TABLE OF CONTENTS

User Manual - Java 2 QED PL

This document is written as a user manual for Java2QEDPL program, which is developed by Kıvanç Muşlu as a senior design project. Java2QEDPL is a translator that translates the input programs (written in Java) to output programs (written in QED-PL).

1. Running

a. *Running Java2QEDPL As a Command Line Tool*

Java2QEDPL can be run as a command line tool using the executable jar file: java2qedpl<version_no>.jar. This file is created via Eclipse and contains only the complied classes. To run Java2QEDPL, the user must give at least 2 input parameters. If the user supplies 'n' parameters to the program, then the first 'n-1' parameters are taken as input files and the last parameter is taken as the output file. Input files must be in the same folder as the executable jar or absolute paths must be given. Output is also generated to the same folder where the executable jar resides (if not given an absolute path) and is overwritten (without any warnings) if the file already exists. User doesn't have to create the output file before running the program, however if user specifies an absolute path for the output file, then s/he must be sure that the directories to reach that path already exists in the system.

The program can be run with the following example pattern.

java -jar java2qedpl.jar <input1> <input2> ... <input_n> <output>

(assuming that the executable is named as java2qedpl.jar)

E.g., java -jar java2qedpl.jar AtomicBoolean.java AtomicBoolean.bpl

or a more complicated one:

java -jar java2qedpl.jar AbstractQueuedSynchronizer.java AbstractOwnable Synchronizer.java LockSupport.java AbstractQueudSynchronizer.bpl.

b. *Calling Java2QEDPL As a Library Method*

Java2QEDPL can also be called as a library method. Include JAVA2QEDPL.jar into your project and build path and call the following method:

Java2QEDPL.translate(String ... args)

Usage of this method is almost the same as the usage of the command line tool. The user must give the method 'n' (String) parameters, where first 'n-1' of them are assumed as input file paths and the last one is assumed to be the output file path.

Example usage:

Java2QEDPL.translate("AtomicBoolean.java", "AtomicBoolean.bpl");

a more complicated one:

Java2QEDPL.translate("AbstractQueuedSynchronizer.java", "AbstractOwnable Synchronizer.java", "LockSupport.java", "AbstractQueudSynchronizer.bpl.");

2. Development: Adding new translations

a. *Detection: Find Where In The AST This Translation Seen*

Java2QEDPL uses JavaCC to generate the complete Java 1.5 grammar. So technically any program that uses this grammar can be translated to QED PL, assuming that it has a correspondence in QED PL. JavaCC uses JavaTreeBuilder (JTB) to generate an Abstract Syntax Tree (AST) from the given program, which is, at the end of the parse process, is represented as a CompilationUnit.

Since Java 1.5 grammar is huge and so the created AST, I have only implemented the parts that is seen in my translations. Thus, if something is not translated at all, there are 2 possibilities: Its visit method (that is to be called by the Visitor pattern) is empty or is implemented partially. So, the first thing that needs to be done to implement this new translation is to decide which visit(s) should be implemented or updated. To decide on this, java_grammer.jtb (which is included in the source file of the project) is a good place to start. This file is taken from JavaCC webpage and contains the complete Java 1.5 grammar to create the AST I mentioned above. So, if you know what expression your code represents in Java grammar, you can easily pinpoint the related visit methods.

b. Implementation: The Visitor Part

After you detect the visit methods that should be implemented or updated, the developer must implement those methods. During the translation process, I take the AST represented with the CompilationUnit (at the top), traverse it with the Visitor pattern (QEDVisitor) and create an internal tree-like structure that represents the whole program internally. This internal representation is something between Java and QED PL and aims to make the translation as easy and clear as possible. The program (which was represented as a CompilationUnit) is represented as a ClassTransformation (MAIN_CLASS_DECLARATION) which is singleton and unique. This virtual class contains all the other classes declared in the program. Classes contain the field members and methods. Methods contain the statements and so on. At the end of this translation, transform method of MAIN_CLASS_DECLARATION is called and the translation is done recursively. Due to this design when something new should be added these are the following possibilities:

- The translation to be added is a statement by itself (i.e., cannot be a part of any other statement): Create an instance of a statement transformation (which already exists or to be implemented), initialize it and set as currentStatement_ in the visitor. Cross your fingers, and the statement will be added to the correct container (method or class).
- The translation to be added is an expression (which can be a part of statement): Create an instance of a transformation expression (super class of all translations), which already exists or to be implemented and add it to the current translation list. Cross your fingers and hope that everything works.
- The above does not work, or the translation is something new: You can always copy the structure of the existing implemented translations. E.g., if you are trying to implement a statement translation and it does not work, look at the translation of IfStatement or WhileStatement to figure out what is missing. Generally the translation structure of the similar expressions are also similar.

c. Implementation: The Internal Part

In my internal data structure all translations are TransformationExpressions (extends this class). When you extend this class, you are required to implemented two methods: transform, which indicates how the translation should be done and getType, which returns the type of the translation to type checking. Before the implementation, one should first think the following: Can this new translation be implemented as a syntactic sugar or using the other other existing translations? If the answer is yes, then you can extend that particular translation (instead of extending TransformationExpression) and only override the transform method so that it does what is required. For more details please see WhileStatementTransformation, DoWhileStatmentTransformation and ForStatementTransformation classes since do ... while and for is implemented as a syntactic sugar for while. The same relation can also be seen in ConstantVariableTransformation, StaticVariableWith AssignmentTransformation, PlainStaticVariableTransformation, and VariableTrans formation hierarchically.

During the implementation of transform method, you might be forced to execute the new translation into a dummy variable and use that dummy variable as an alias to the actual translation (alias unrolling). This is required do to the difference in Java and QED PL. QED PL is developed to prove concurrent algorithms and assumes every sentence written in QED PL is atomic. However in reality this is not the case. In Java, cascaded method calls, variable accesses, global accesses inside ifs, and much more are example for this case. Also in QED PL, if there is method call in a line, it should be a line itself (i.e., method calls cannot be used as a part of a line). Because of these, the above strategy (alias unrolling) must be used sometimes. However, the problem is that, if you come to the point where you should translate that expression and it were to be alias unrolled it is too late in the translation. Consider the following example:

Java Code: `counter = 3 + getCounter();`

Wrongly translated QED code (attempt to translate at the moment of the method call translation): `counter := 3 + call dummy := getCounter(); dummy;`

As you see from the example, alias unrolling of getCounter() must be done before the translation of the sentence: ‘counter = 3 + getCounter()’.

To do this, I have used the listener-notifier pattern and events. There is an event (interface) called UpdateEvent and a listener (UpdateListener). For now, only the MethodTransformations implement the UpdateListener (it was sufficient to do the alias unrolling before the method’s translation), however any class can implement this interface. Before the translation of the methods, (as a listener) it invokes the doEvent() (method that comes from UpdateEvent) method of each event so that this alias unrolling can be done before the actual translation. During the doEvent() method the general structure is as follows:

- Prepare what the translation needs the write before the actual translation (which should be somehow assigned to an alias).
- Adds this translation as a pre-translation or pre-initialization (pre-initializations are written before the pre-translations) for the container of the translation. Container represents the parent in my tree-like structure. E.g., in the above example the container of getCounter() is the whole expression.
- Notice that at this moment nothing in the method (including the above expression) is translated yet. So when the above expression to be translated it will first write the pre-translation coming from getCounter(). (During the translation first pre-initialization is written, then pre-translation is written and finally the translation itself is written.)

With this new formation the above translation would be as:

QED Code:
```
call dummy := getCounter();
counter := 3 + dummy;
```
which is correct.

Lots of examples in the source code can be found and the pattern is actually quite powerful and easy to implement. Some examples: TernaryOperator
Transformation, MethodCallTransformation, and VariableAccessTransformation.

3. Non-trivial Translations

a. Array Access Transformation

Arrays are defined as following in QED PL as follows:

```
record Array<Type> {
        elements: [int]Type;
        length: int;    }
```

So an array access is actually getting the variable from the 'elements' map or setting the variable into the map. During array access an index must be given. It is a fact that this index must be grater than 0, and less than the array length. Also this index may be a complex expression. So, during the translation, I first alias-unroll the index to a dummy integer variable. If the array access translation is a left hand side (LHS) of the assignment, that is all, the assignment is done. However, if the array access translation is a right hand side (RHS) of the assignment, then I also alias-unroll the whole array access expression to a dummy variable of type 'Type'. Here are two example translations:

Java Code - LHS	**QED PL Translation**

array[index] := 5;
// assuming that array is an integer array.
	var dummy: int;
	dummy := index; // alias-unrolling for index.
	// assertion that represents the above assumption.
	assert 0 <= dummy && dummy < array.length;
	array.elements[dummy] := 5;

Java Code - RHS	**QED PL Translation**

counter = array[index];
// assuming that counter is of type integer.
	var dummy: int; // for index.
	var dummy2: int; // for array access.
	dummy := index; // alias-unrolling for index.
	// assertion that represents the above assumption.
	assert 0 <= dummy && dummy < array.length;
	// alias-unrolling for array access.
	dummy2 := array.elements[dummy];
	counter := dummy2;

b. *Array Initialization Transformation*

All object creations (heap allocation) must be translated separately in QED PL. This is because the 'new' keyword in Java normally represents two things: Heap allocation and calling the related constructor of the newly created object. For the array initializations this is a little different: One must allocate the array and then set its length to the given value. Moreover, since the allocation can be used in an expression, we also have to alias-unroll the whole translation. Here is an example:

Java Code	**QED PL Translation**

array = new int [10];	var dummy: Array;
	dummy := New Array; // alias-unrolling for array initialization.
	dummy.length := 10;
	array := dummy;

c. *Assignment as Expression Transformation*

In Java it is possible to use assignments as expressions (especially in if clauses), where as in QED PL we don't have such a structure. Thus, these expressions should be translation as follow: First translate the assignment, then use assigned variable in the expression. This time there is no alias unrolling, however still the assignment part should be executed before the expression, so event model is used in the translation. Here is an example:

Java Code	**QED PL Translation**

number = (dummy = 2) + 3;	dummy := 2; // assignment is done first.
	number := dummy + 3;

d. Class Transformation

Since QED PL is not object oriented, classes are represented as records. However there are some limitations in QED PL. For example, one cannot execute code inside a record, only declare variables (such as field assignments). Also since QED PL is procedural, I have to pass ‘this’ as an argument to each class method. For detailed information about Class Translations please see Javadoc information, and here is a detailed example.

Java Code	**QED PL Translation**

class Counter {
    private static int dummy = 3;
    private static final int funny = 5;
    private int buzzy = 2;
    private int count;
 	
    public Counter(int c) {
        count = c;
    }
    public static int getDummy() {
        return dummy;
    }
}
	// Class name prefixed to static variables so there will not be more than one declaration of the same variable.
	var Counter_dummy: int;
	// static variables are declared and assigned before the record declaration.
	Counter_dummy := 3;
	// static final variables are translated to const. uniques.
	const unique Counter_funny;
	axiom (Counter_funny == 5);
	// actual class representation as record.
	record Counter {
	    buzzy: int;
	    count: int;
	}
	// methods are translated as procedures, first parameter for class methods are always ‘this’. All classes have default constructors (to assign field members).
	procedure Counter_Counter(this: Counter) {
	    buzzy := 2;
	}
	// in QED PL procedures cannot be overloaded, so a new name is given to the actual constructor.
	procedure Counter_Counter2(this: Counter, c: int) {
	    // All constructors except the first one call the default constructor to initialize the field members.
	    call Counter_Counter(this);
	    // count can only be accessed through this.
	    this.count := c;
	}
	// Static methods does not take ‘this’ as parameter.
	procedure Counter_getDummy() returns (result: int) {
	    // static member is access as a normal variable.
	    result := Counter_dummy;
	}

e. Compound Assignment Transformation

Compounds assignments (+=, -=, ++, --) are not defined in QED PL. Moreover, in reality they are just shorthand notations used in Java. So, I just translate them to their actual representation. Note: ++, and -- are also taken as compound assignments since they are parsed as syntactic sugar to other compound assignments while traversing the AST. (++ as += with argument 1, e.g.) Here is an example:

Java Code	**QED PL Translation**
number += 3;	number := number + 3;

f. Do While Statement Transformation

Do ... while statements are nothing but a syntactic sugar to while statements. Since QED PL does not have 'do ... while', in my translation they are also translated as while statements. Here is an example:

Java Code	**QED PL Translation**

```
do {
        counter ++;
} while (counter < 10);
// Assume that counter is local.
```
	```
	// First iteration is written to the top of the while no matter what.
	counter := counter + 1;
	while (counter < 10) {
	        counter := counter + 1;
	}
	```

g. Exception Statement Transformation

Exceptions are defined as const uniques in QED PL. So whenever an exception is seen, it is added to the MAIN_CLASS_DECLARATION so that it will be defined at the beginning of the output. Also QED PL doesn't have a type for String, so the exception message is ignored and that constant unique is thrown every time an instance of that exception is thrown. Here is an example:

Java Code	**QED PL Translation**

```
if (a == null)
        throw new NullPointerException('a is null');
// Assume a is local.
```
	```
	const unique NullPointerException: Exception;
	if (a == null) {
	        throw NullPointerException;
	}
	```

h. For Statement Transformation

For statements are nothing but a syntactic sugar to while statements. Since QED PL does not have 'for', in my translation they are also translated as while statements. Here is an example:

Java Code	**QED PL Translation**

```
for (int i = 0;  i < 10; i++) {
        counter ++;
}
// Assume that counter is local.
```
	```
	// Initialization of for is done before the while statement.
	var i; int;
	i := 0;
	while (i < 10) {
	        counter := counter + 1;
	// Update part is executed at the end of while, each iteration.
	        i := i + 1;
	}
	```

i. Method Call Transformation

Method calls must be alias-unrolled since in QED PL, you cannot use method calls inside statements. It is also important to figure out the first parameter to pass to the method call. This is done in runtime. If the method is a class method (instance method) than the first parameter is also the reference that calls the method (if none there it is ‘this’) (this was done to model the object oriented programming). If the method call is for a static method, then there is no parameter. Here is an example:

Java Code	**QED PL Translation**

prev = node.prev();
// prev() is defined in Node class and returns a Node.
	var dummy: Node;
	dummy := Node_prev(node); // The calling reference is node.
	prev := dummy;

j. Method Transformation

Methods are translated as procedures in QED PL. All methods receive the class name, that they are declared in, as prefix to their procedure declaration to prevent the declaration of methods with the same name. Overloaded methods are also given new names (an extra integer at the end of the method) to prevent the same problem. Static methods does not take an extra parameter, however instance methods (non-static class methods) take a reference to the class (as ‘this’) to simulate object oriented programming in java. Methods with ‘package’ and ‘public’ modifiers are translated as ‘public’ procedures where as methods with ‘protected’ and ‘private’ modifiers are translated as ‘private’ procedures. In QED PL, procedures can return multiple values and these values must be declared in the method signature. On the other hand in Java, methods can return at most one value (or can be void). So, a default return value, called ‘functionResult’, is added to the procedure signature for methods that return a value.

Also in QED PL, one can declared the methods as atomic (with the annotation of {:isatomic true}), where as this is not possible in Java by default. On the other hand, some Java programs, especially the low-level standard library code, can use native Java methods that are implemented with C++ and have only Java method prototype. They have no method implementation, and since they are implemented with C++, they are either atomic using the hardware support, or they are very hardware specific. So, a user might need to annotate this native methods with a corresponding QED PL code, that tells how to translated this method to QED PL (since they don’t have a real implementation). To solve this problem, I have defined a Java annotation called JAVA2QEDPL. It’s constructor takes a string array which represents the corresponding procedure translation for that methods. So, whenever a method is seen with the ‘JAVA2QEDPL’ annotation, then the whole body of the method is ignored and the implementation given in the annotation’s constructor is written as the method translation. The user can also use JAVA2QEDPL.THIS, and JAVA2QEDPL.RESULT to access the ‘this’ variable that represents the class instance for non-static class methods and ‘functionResult’ that represents the return value. The annotation must be written purely in QED PL and it is not checked (syntactically). Also annotated methods are automatically translated as atomic procedures for now. Here is a detailed example:

Java Code	**QED PL Translation**

// Assume that all methods are declared in Thread class.
// id is a field of Thread.
public int getID() { return id; }
	procedure Thread_getID(this: Thread) returns (functionResult: int) { // ‘this’ is passed as an argument.
	    functionResult := this.id;
	    return; // id is accessed through this.
	}

`// I know silly but for example :)`
`private int getID(int what) {`
`        return id;`
`}`
`// Assume currentThread exists, static.`
`public static Thread currentThread() {`
`        return currentThread;`
`}`
`// Assume Unsafe.sleep() is native.`
`@JAVA2QEDPL( {`
`"assume true;" })`
`public static int sleep() {`
`        Unsafe.sleep(1000);`
`}`
	`// Name of the method is changed with postfix '_2'`
	`procedure {:ispublic false} Thread_getID_2(this: Thread) returns (functionResult: int) {`
	`        functionResult := this.id;    return;`
	`}`
	`procedure Thread_currentThread() returns (function Result: Thread) { // Note that there is no 'this'.`
	`        functionResult := Thread_currentThread;`
	`} // currentThread is accessed directly.`
	`// JAVA2QEDPL.RESULT is figured using reflection and the whole translation is copied from the annotation.`
	`procedure {:isatomic true} Thread_sleep() {`
	`        assume true;`
	`}`

k. Return Statement Transformation

Since 'return' cannot take an expression in QED PL, and can in Java, it is translated as an assignment to 'functionResult' and plain 'return'. Here is an example:

Java Code	**QED PL Translation**
`return 3;`	`functionResult := 3;    return;`

l. Short Circuit Transformations: '&&' and '\|\|'

In Java (and in many programming languages) '&&' and '||' have special meaning. If the first operand is true (for ||) or false (for &&), then the second operand is not evaluated. In other words, the boolean operations are done until the result can change. E.g., for cascaded 'or's , the result is found whenever the first 'true' evaluation is detected. The remaining expression are never evaluated. This is something important since some of the expressions may have side-effect or might lead to exceptions if they were to evaluated. Consider the following code: if (node != null && node.prev != null) Here, if 'node' is really 'null' and if we would have evaluated all the expressions then the above expression were forced to throw 'NullPointerException'. Thus this statements should be translated with extra care (since we don't have short circuit expressions in QED PL). I translate them into plan if statements however each if statement is executed depending the result of the current expression's evaluation. E.g., for '&&' if one of the expressions are evaluated as 'false', then the remaining 'if' statements also evaluate to 'false' and the remaining expressions are not executed. Since short circuit translations can be used as plain expression (especially they are used a lot in if statements), their translation must be alias-unrolled to a dummy variable. Here is an example:

Java Code	**QED PL Translation**

`if (node != null && node.prev() == n) {`
`        // do something.`
`} // Assume node and n are local.`
	`var dummy: bool;    var dummy2: Node;`
	`dummy := node != null; // evaluate 1st expression.`
	`if (dummy) { // If the 1st expression is true,`
	`        call dummy2 := Node_prev(node);`
	`        dummy := dummy2 == n;`
	`}`
	`if (dummy) { // If both expressions are true, // do.}`

m. Variable With Assignment Transformations: static or non-static (plain)

In Java, the programmer can define a variable and assign it to a value at the same time, however this is not possible in QED PL. So, the variable declaration and the assignment is done in two steps. Static variable assignments are done after the declaration and instance variable assignments are done in the default constructor (as in Java, internally). Here is an example:

Java Code	**QED PL Translation**

// Assume all variables are declared in class MyClass.
public static int number = 3;
public int no = 5;
	var MyClass_number: int;
	MyClass_number := 3;
	record MyClass { no: int; }
	procedure MyClass_MyClass (this: MyClass) {
	    this.no := 5; // default constructor initialization.
	}

n. Ternary Operator Transformation

In Java '... ? ... : ....' is called the ternary operator. This operator is a shorthand for an 'if ... else ...' statement. In QED PL, there is no ternary operator. So the translation is done by translating the ternary operator to the corresponding 'if ... else ...' statement. To do this, I first decide the result type of the ternary operator, which is defined as the super-class of the result types of its operands. After that the ternary operator translation is alias-unrolled to a dummy variable with that type. Here is an example:

Java Code	**QED PL Translation**

int counter = node == null ? 2 : 3;
// Assume node is local.
	var counter: int;
	var dummy: int; // for ternary operator.
	var dummy2: bool; // for ternary operator condition.
	dummy2 := node == null;
	if (dummy2) { dummy := 2; } else { dummy := 3; }
	counter := dummy;

o. Try ... Finally Transformation

In QED PL, there is no 'finally' keyword. In Java 'finally' represents a code block that should be executed whatever happens (in case of exception or normal execution). It is generally used to release locks, close connections, cleanup, etc. In QED PL, the last exception thrown is preserved in a variable called 'ex'. Using this, I have implemented 'try ... finally' translation as 'try ... catch' in QED. Inside of try is translated, the code in the finally block is translated and put in the 'catch' block of the translation (if exception happens) and after the 'catch' block (if normal execution). In the catch block, 'ex' is re-thrown to complete the translation. Here is an example:

Java Code	**QED PL Translation**

try {
    // do a
} finally {
    // do b
	try {
	    // do a
	} catch { // catches all possible exceptions.
	    // do b.
	    throw ex; // Re-throw the last exception.
	}
	// do b. // Represents normal execution.

p. *Type Allocation Transformation*

Type allocation (‘new’ keyword) normally represents two things in Java: Allocating the heap for the new variable and calling the constructor of the new variable. Thus, in QED PL, type allocations are translated as two statement executions. Also, since in Java type allocation can be used as an expression, the translation must be alias-unrolled. Here is an example:

Java Code	**QED PL Translation**

Node n = new Node(data);
// Assume data is local.
	var dummy: Node; var n: Node;
	dummy := New Node; // Allocation.
	call Node_Node_2(dummy, data); // Constructor call.
	n := dummy; // alias-unrolling.

q. *Variable Access Transformation*

QED assumes every line in the source code (written with QED PL) as atomic expressions. However, in Java, programmer can use a global access within another expression. In reality, this is also translated as first getting the value of the global access and then using it in the expression by the Java compiler. However in QED PL, this has to be done explicitly. To do this, I first decide weather the variable access is a local access or a global access. This is done by the partial implementation of the scoping rules in Java. If the variable access is a global access, then I follow the cascaded accessed variable and decide its type. Then, the whole access translation is alias-unrolled to a dummy variable created in this type. Alias-unrolling must be done, since the global access can be used as a part of an expression. For detailed information on how I decide a variable access is global or local and how I follow the cascaded access, please see the documentation of VariableAccessTransformation.java. Here is an example:

Java Code	**QED PL Translation**

// Assume value is a member of class C.
value := value + 1;
	var dummy: int;
	dummy := this.value;
	// Notice that the variable is not alias-unrolled if LHS.
	this.value := dummy + 1;

r. *While Statement Transformation*

Normally while statements are translated directly to QED PL without any problems. However since the condition of the while must be alias-unrolled to another variable (it might be a global access, etc.) and this condition should be evaluated in each iteration, at the end of the while statement translation, the pre-translation coming from condition variable is attached for each iteration. Since for statement and do ... while statement is translated as a syntactic sugar to while statement (they extend WhileStatementTransformation), they have the same attitude. Here is an example:

Java Code	**QED PL Translation**

while (node.prev != null) {
        node := node.prev;
}
	var dummy: bool; // alias-unrolling for while condition.
	dummy := node.prev != null; // node.prev is a global access.
	while (dummy) {
	        node := node.prev;
	        dummy := node.prev != null; // This must be added.
	}