View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by MURAL - Maynooth University Research Archive Library

Ensuring behavioural equivalence in test-driven porting

Mark Hennessy
Computer Science Dept.
National University of Ireland
Maynooth, Co. Kildare, Ireland
<markh@cs.nuim.ie>

James F. Power
Computer Science Dept.
National University of Ireland
Maynooth, Co. Kildare, Ireland
<jpower@cs.nuim.ie>

Abstract

In this paper we present a test-driven approach to porting code from one object-oriented language to another. We derive an order for the porting of the code, along with a testing strategy to verify the behaviour of the ported system at intra and inter-class level. We utilise the recently defined methodology for porting C++ applications, eXtreme porting, as a framework for porting. This defines a systematic routine based upon porting and unit-testing classes in turn. We augment this approach by using Object Relation Diagrams to define an order for porting that minimises class stubbing. Since our strategy is class-oriented and test-driven, we can ensure the structural equivalence of the ported system, along with the limited behavioural equivalence of each class. In order to extend this to integration-level equivalence, we exploit aspect-oriented programming to generate UML sequence diagrams, and we present a technique to compare such automatically-generated diagrams for equivalence. We demonstrate and evaluate our approach using a case study that involves porting an application from C++ to Java.

1 Introduction

Porting is defined as the meaning-preserving transformation from one programming language to another. When programming language features become obsolete or another programming paradigm becomes widely used then it is often desirable to port useful programs and systems to a new programming language. Some examples of this include the original lexical analyser generator for C, *lex*, which has been ported to many other programming languages including Java, Python and Perl. Another example is the unit testing tool for Java, JUnit [2], which has been ported to C++ as CppUnit.

The porting of code does not automatically guarantee that the ported system will work exactly as the original. Errors may enter the ported code through a lack of comprehension of the features of the system to be ported, through subtle differences between the programming languages or through errors committed by the programmer. The ported system must be tested rigorously to ensure that errors have not entered the ported code.

In this paper we describe the porting of the C++ analysis tool, keystone [6], from C++ to Java using a testing strategy that ensured the correctness of the ported code. Our work extends the *eXtreme Porting* approach outlined by Varma *et al.* for porting C/C++ applications [30]. This process operates by porting one class at a time and then unit testing to ensure correctness.

When porting from one object-oriented language to another, we discovered that it is not altogether obvious as to what order the individual classes should be ported such that the use of stubs when unit testing the ported classes is kept to a minimum. In this paper we define an order for the porting of classes from one object-oriented language to another using an *Object Relation Diagram* (ORD). Previous work on ORDs used them to define an order for inter-class testing, and we extend this to test-driven porting.

One feature of the keystone system is that it

Accepted for the CASCON 2006: Dublin Symposium
Dublin, Ireland, October 17, 2006

has a well-defined separation between its front-end, which parses C++, and its back-end which performs semantic analysis and outputs a summary of the program. We exploited this separation by using a record-and-replay testing harness, a technique commonly used in problem domains that share a well defined border between front-end and back-end, such as migration in back-end web services or the transformation of the underlying code of a GUI system.

Since the keystone system can be used as an API as well as a stand-alone tool, its output is merely a summary of the semantic information it collects. Thus we took the view that black-box testing would not provide satisfactory assurance that the ported version of keystone was working correctly. In order to verify the internal behaviour of the ported system we conduct white-box tests on crucial elements. The white-box tests are implemented using Aspect Oriented Programming (AOP). We exploit AOP by weaving similar aspects through both systems that have the effect of dynamically generating a UML sequence diagram for each test case. A comparison of sequence diagrams using Hirschberg's algorithm [11] on a case-by-case basis then provides assurance that the behaviour is identical in each case.

The remainder of this paper is structured as follows. In Section 2 we review the background and related work for the concepts presented in this paper. In Section 3 we describe our method for producing an order for our porting. Section 4 describes our testing methodology for both black-box and white-box testing. Our approach for generating sequence diagrams on the fly is outlined. The results presented in Section 5 outline the effectiveness of the testing strategy. A comparison of the performances between the two systems are also provided. Finally, Section 6 concludes the paper.

2 Background

In this section we briefly review the main concepts underlying porting and particularly in testing the ported system. We present an overview of the testing techniques that were used during our work and we discuss our strategy for generating sequence diagrams to aid in the testing process.

2.1 The keystone system

The system that we wish to port is keystone, a parser and static analysis tool for ISO C++ programs [6]. The system can be used either as a black-box tool, producing a summary of an ISO C++ program, or as an API that can provide access to the ISO C++ program being processed, allowing the programmer to conduct their own analysis.

The keystone system is written entirely in ISO C++ and consists of 40 classes and roughly 23,000 lines of code with many inter-class dependencies and inheritance hierarchies. The front-end of keystone consists of a tightly coupled lexical analyser and backtracking bottom-up parser. The back-end of keystone consists of a set of semantic routines called by the parser to maintain a symbol table, and to represent the scope and type information contained in the input program.

As we plan to implement our own front-end in Java based on a separate experimental *Generalised LR* parser, our goal was to port the keystone back-end only. By decoupling the front-end and back-end, we can maintain the original front-end of the ported system, thus ensuring that the original test cases can be used with the ported version of the system. This approach is often found in web-based or GUI-based systems, which are typically designed to decouple front-end and back-ends to maximise portability.

2.2 Test-driven Porting

The area of porting is concerned with a meaning-preserving transformation from one software architecture to another. This transformation can encompass formal techniques such as tree transformations from an AST or less formal techniques such as partial generation of code from CASE tools and manual porting from source code. While some automation of the porting process is possible, this is highly dependent on the source and target languages being used. Any part of the process that is subject to manual intervention must also be sub-

ject to extensive testing to ensure correctness.

Testing is the most crucial phase in the development of any software system. Recent studies have shown that up to 60% of a project's life-cycle can be taken up by testing concerns [21]. eXtreme Programming espouses a philosophy of rapid development along with well designed test-cases for unit testing under the motto "*Test early and test often*" [1]. This philosophy is supported in practice by tools for testing such as JUnit for Java [2]. The goal of early testing is to have unit tests written before any code. This ensures that the design of the system can be tested as soon as the class code is implemented.

Recent work by Varma *et al.* defines a methodology for porting C/C++ applications called *eXtreme Porting* [30]. The main tenet of this methodology is based around a rigorous testing regime that involves unit testing of every ported class followed by integration testing of related clusters within the system. Finally an acceptance test must be passed before the system can be considered successfully ported. However this porting method does not contain an explicit procedure for the order of porting the classes.

When two or more classes that have an interaction with each other are implemented, it is desirable to test their interaction through integration testing. Where a class interacts with another, as yet, unimplemented class, then class *stubs* need to be used to simulate the functionality of the unimplemented class [20]. Stubs usually implement a subset of the class functionality but may have to replicate the complete class functionality in extreme cases. Thus, where possible, the use of stubs should be kept to a minimum due to the costs involved in their construction.

The unit tests and integration tests aim to ensure that each class has been tested but they may not always guarantee that the overall system is free of bugs. System tests must be conducted on the software system as a whole to verify that the design and requirements have been implemented correctly according to the specification. System tests are most commonly divided into black-box and white-box tests [23]. Black-box, or specification-based testing, seeks to test a program without any reference to the internal structure of the program. A test-case is executed through the system and the output is compared to an expected output derived from an oracle. White-box testing, or structural-testing, is concerned not only with the output of a given test-case but also ensuring that correct areas of code have been executed during the testing.

2.3 Porting Strategy

The porting strategy described in the remainder of this paper is based on our experiences of porting the back-end of the keystone system, written in C++, to a functionally-equivalent version written in Java, which we refer to as jKeystone. An important facet of the porting process was that it was *test-based* and *test-driven* in order to ensure the correctness of the ported system.

The overall structure of the porting process was as follows:

Step 1 The keystone front- and back-end were decoupled. Fortunately, there is a clear distinction between these keystone phases, and the classes in the back-end are easily identified.

Step 2 The back-end keystone classes were then ported to Java. This consisted of three steps:

- **Step 2a** Dependencies on external libraries were identified. keystone has relatively few library dependencies, mostly classes in the C++ Standard Template Library, and it was not difficult to identify classes with similar functionality in the Java class library.
- **Step 2b** The jKeystone class hierarchy was generated. The class hierarchy of keystone was reverse engineered using an in-house tool, and the corresponding class hierarchy of jKeystone was generated. This was then edited by hand to ensure that parameter and attribute types were correctly represented. No method-body code was ported automatically in this step.

Step 2c Each class in turn was then ported and unit-tested, in the vein of the *eXtreme Porting* approach.

Step 3 keystone's front-end was modified so that when run over a test case, it generated a Java test harness for that test case. These were then used to perform black-box system tests on jKeystone.

Step 4 Aspect Oriented Programming was used to weave similar tracing aspects across both keystone and jKeystone. For each test case, the aspected version of keystone and jKeystone produced a sequence diagram, containing the objects and interactions involved in processing that test case. These were then compared to verify the behaviour of jKeystone.

Steps 1, 2a and 2b are mostly routine, particularly when porting from relatively similar languages such as C++ and Java. While Step 2c is based on the approach outlined by Varma *et al.* [30], they do not address the important issue of what order the classes should be ported in; our solution to this is described in Section 3. The system testing using a test harness is similar to that used in GUI testing, but we extend this to apply to white-box testing using sequence diagrams. Both kinds of system testing are described in Section 4.

3 ORD-based porting

In this section we outline our algorithm for ORD based porting of a system. Our method for deriving the port order is outlined along with an example.

Since C++ is a relatively difficult language to parse and analyse, there is a relative scarcity of automated tools that support porting. When moving from C++ to Java, the class and method signatures can be generated automatically, but the overwhelming majority of the code must be ported by hand. To ensure that any bugs that inadvertently entered the system during porting were caught and eliminated, the principles of eXtreme programming "test often and test early", were applied.

3.1 Object Relation Diagrams

The eXtreme Porting strategy does not contain an explicit procedure for the order of porting the C++ classes. It is desirable to port the classes from C++ to Java in an order that the number of class stubs needed during testing is kept to a minimum. A key insight in this paper is that the porting of classes from one object-oriented language to another bears a similarity to that of defining an order for inter-class testing.

There are many examples of work in the area of defining an order for testing object-oriented systems that make use of a type of graph known as an *Object Relation Diagram* (ORD) [15, 28, 5, 22]. The ORD can be loosely compared to a UML class diagram as it features classes as nodes and the relationship between classes as edges. The three edges typically represented within an ORD are *Associations*, *Composition* and *Inheritance*. *Associations* themselves can be further decomposed into simple aggregations, dependencies and associations.

Once the ORD has been constructed, it is possible to assign costs to the edges and nodes, and use these to define a testing order, so that the class with the smallest number of dependencies can be tested first. To this end, it is usual to define an ORD *cost model* that determines the order in which classes can be tested so as to minimise the need for class stubs. We observe that the same problem exists in porting, specifically in regard to the order in which to port the classes within the system. By porting the classes according to a costed ORD, we can dramatically reduce the amount of class stubs needed during the unit testing phase.

Malloy *et al.* have enhanced the basic ORD cost assignment by developing a *parameterised* cost model [16]. Such a cost model allows the ORD to be configured and changed easily, so that it can be fine tuned for a specific application. With the parameterised cost model, each type of edge within the ORD is given a specific weight and all the edges between two classes are merged and their weights are summed. The ORD is divided into strongly connected components (SCC) and the edge with the smallest weight is removed. This process is repeated un-

til all the SCCs contain only 1 node. Finally the ORD is reverse topologically sorted to give an inter-class test order.

Milanova *et al.* have extended the idea of the ORD further by developing the ExtORD which is an ORD created at the precision of the statement level [22]. Multiple edges of each kind of dependence between the class nodes are utilised to give this precision. The level of precision in this strategy is unnecessary for our porting strategy but this approach is useful in identifying coverage of statements that trigger inter-class dependencies.

3.2 A cost model for porting

We have devised an algorithm to cost an ORD specifically for the purpose of porting. Our algorithm is outlined in Figure 1.

The algorithm works by applying the cost model to all root classes of inheritance groups initially, as outlined in Step 1 of Figure 1. Root classes are identified as those that have an incoming inheritance edge but no outgoing inheritance edge. The weight calculation, described in Step 3 of Figure 1, then calculates the weight of the current node. It adds the current weight of the node to the *sum* of all the incoming Composition and all outgoing Association and Inheritance weights. The sum function then recursively calculates the cost of every direct child node from the current node if an incoming Inheritance edge exists. Finally, Step 2 of the algorithm calculates the cost of all of the remaining edges within the ORD.

In order to demonstrate the operation of the cost assignment algorithm shown in Figure 1, we apply it to the ORD shown in Figure 2(a). This ORD was originally presented by Briand *et al.* [5]; we use it here to facilitate comparison of our approach with theirs. The ORD in Figure 2(a) has 8 classes, labelled A through G, with inheritance, association and aggregation edges labelled I, As and Ag respectively. In this example there are 3 inheritance edges, demonstrating both single and multiple inheritance, 3 aggregation edges and 11 association edges.

By following the algorithm outlined in Figure 1 we can cost our ORD as follows:

- We select a weighting for each of the edge types of $I = 5$, $As = 35$ and $Ag = 60$. These weights are based on Malloy *et al.* and adjusted heuristically, noting that the algorithm we use automatically assigns a relatively high priority to inheritance.
- Following Step 1, the root classes in the inheritance hierarchy are A, B and H; by applying step 3a we assign a weight to these

Initialisation: Build the ORD labelling edges as Inheritance, Association and Aggregation.
Assign weights to each of the *edges* according to our cost model.
Then propagate weights to the nodes by performing Steps 1 and 2 below.

Step 1: Identify the root nodes of Inheritance hierarchies by choosing each node with with one or more *incoming* Inheritance edge and no *outgoing* Inheritance edges.
Perform *Step 3* for each Node identified.

Step 2: Identify the Nodes that have no *incoming* or *outgoing* inheritance hierarchies.
Perform *Step 3* for each Node identified.

Step 3: Calculate the weight for the current node as follows:

Step 3a: The weight is calculated as the current weight added to the sum of the *incoming* Aggregations, the sum of the *outgoing* Associations, the sum of the *outgoing* Inheritances and the value of any weight parameters passed.

Step 3b: For each child of the current Node, identified as an *incoming* Inheritance edge of the current Node, perform *Step 3a* passing the child Node and weight of the current Node as parameters.

Figure 1: The algorithm to determine the porting order for classes. *When run over an ORD this algorithm assigns a weight to each class, and the class with the lowest weight is ported first.*

classes based on their edges.

- We then perform Step 3b, propagating these weightings down the inheritance hierarchy in a depth-first manner, assigning weights to nodes D and G.
- Finally we apply Step 2 to those classes not so far covered, and assign weights to classes C, E and F.

The final values assigned to each class are given in Figure 2(b). This defines a testing/porting order where we start with the smallest weight and work upwards. In this example, just two classes, C and D need to be stubbed, and only the first three classes to be ported have direct dependencies on these stubs.

3.3 Porting keystone classes

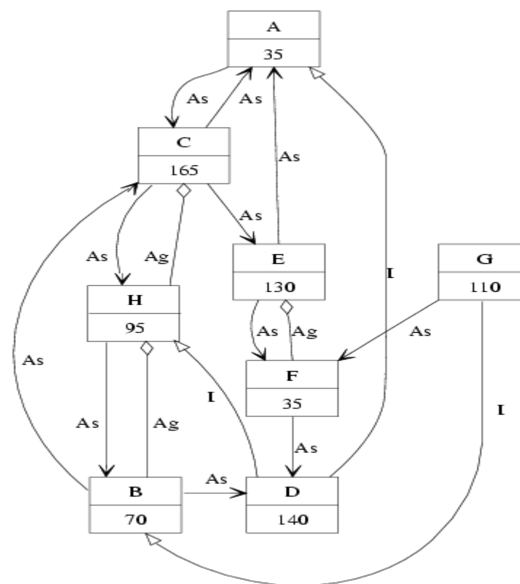
Applying our algorithm to the ORD for keystone we can derive a porting order that involves the creation of only three stubs for unit testing during the porting. The keystone ORD can be seen in Figure 3. In this ORD, the classes are numbered in the order that they were ported. The three stubs that are needed are the classes numbered 13, 14 and 37 respectively. The next section describes our system testing strategy for the newly ported system.

4 System Testing

The use of the ORD in our porting strategy ensures that as we port, the *architecture* of keystone is preserved. However it is also essential that the *behaviour* of the original system remains identical in the ported version. The unit tests provide some guarantee, but further testing is necessary to ensure that the system as a whole behaves as expected. In this section we describe the testing of the ported jKeystone system using back-box and white-box tests.

4.1 The System Test Suites

To test the system in order to ensure that jKeystone was a correct port of keystone, two separate test suites are used:



(a) An ORD with eight classes.

	Class	Weight	Stubs
1	A	35	C
2	F	35	D
3	B	70	C,D
4	H	95	None
5	G	110	None
6	E	130	None
7	D	140	None
8	C	165	None

(b) The porting order for the ORD

Figure 2: An example of using an ORD to define porting order. *Here, the edges have been weighted using the values I=5, As=35 and Ag=60. The table shows the order in which the classes would be ported and the stubs on which they depend.*

GCC The popular, open source GNU compiler collection *gcc* includes a C++ compiler that implements ISO C++ along with a large test-suite for all of the various languages accepted by the compiler. The C++ specific test-suite in the *g++.dg* test-suite distributed with *gcc* version 4.0.0 contains roughly 1,800 C++ programs and contains both positive and negative test-cases.

This is an *implementation-based* test-suite,

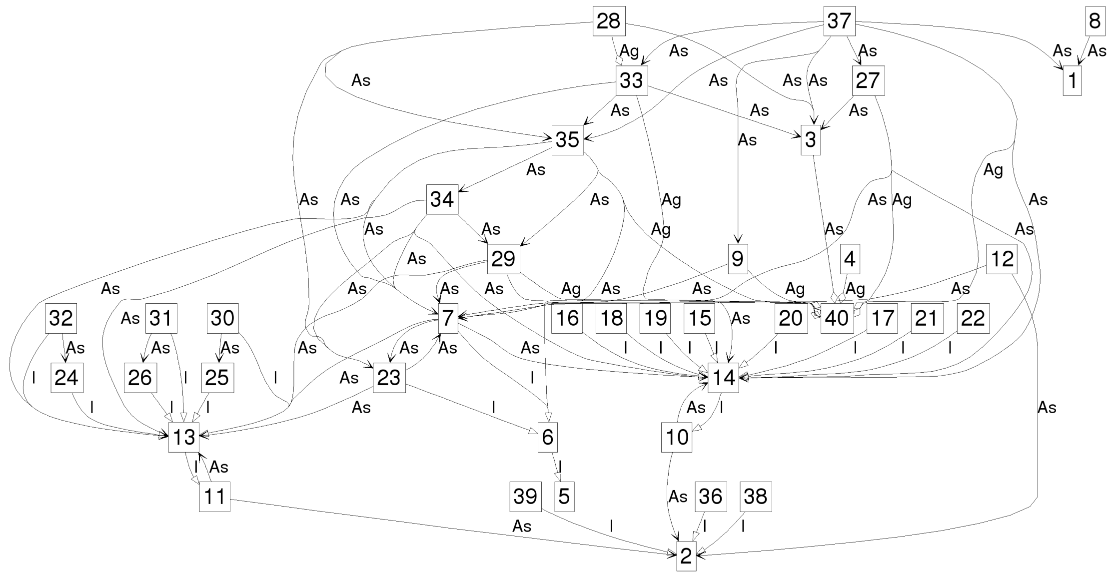


Figure 3: The Object Relation Diagram for keystone. *In this diagram the nodes represent classes and the edges represent inheritance (I), association (As) and aggregation (Ag) relationships between the classes. The number at each node indicates the porting order, with 1 indicating the first class to be ported.*

in that it was assembled to test all of the features of the compiler and it has been augmented many times with new test-cases as new features have been added or as bugs have been discovered.

DDJ The other approach to creating a test-suite is to generate test-cases directly from a specification that cover all aspects of the language. The *DDJ* test-suite, is a *specification-based* test-suite that has been developed to test compliance of different parsers to the ISO standard [7, 17]. This suite consists of 440 individual test-case and each test-case has been derived directly from a clause within the ISO standard [13].

While some research has been done on automatic generation of test suites for grammar-based software, this has been shown to be less effective for a language with complex semantics such as C++ [18]. Our own work on test-suite reduction demonstrated convincingly that adequate testing of grammar-based software required a comprehensive test suite with good coverage of both front- end back-ends [9, 10], such as is provided by the test suites selected here.

The *negative* test-cases in the *gcc* suite were not used for our system tests, reducing the total size of the test-suite to 1321 test cases. Negative test cases are those that are not valid C++ programs, and are designed to ensure that the error handling of *gcc* is robust and correct. However, the use of the record and playback approach described below meant that a negative test case would not trigger the generation of driver code for jKeystone, and hence could not be used to determine the correctness of the ported code. This was not of concern in this project, but may be an issue for different kinds of application.

4.2 Record and playback

As outlined in Section 3, only the keystone back-end was ported. Thus, in order to complete the system tests for the ported system it was necessary to emulate the operation of the keystone front-end, a parser generated by the tool *btyacc*. The parser front-end is respon-

sible for parsing the input file and hence the calling of the corresponding semantic actions in the back-end.

To facilitate this emulation, a *record and playback* feature for the operation of the C++ front-end was implemented. This feature is used heavily in testing other systems such as GUIs [21] and network firewalls [12] and operates by recording the correct operation of a test-case in a manner that can then be used automatically at some point in the future, without any human input.

Record and playback was achieved by instrumenting the keystone parser front-end to generate Java code as it parsed its C++ input. Since the keystone parser was generated using the parser generator *btyacc* it was contained in a monolithic file. While this usually causes problems for modular development, in this case it proved an advantage, facilitating the instrumentation process. The generated Java code produced from the instrumentation acts as a driver for the test-case, calling the back-end routines in jKeystone.

Figure 4 presents an overview of the system testing process. The input test case, a C++ program from one of the test suites, is depicted on the left of the figure. When used as input to keystone this generates the normal test results, which are stored for comparison later. It also causes the instrumented keystone front-end to generate Java driver code that is specific to that test case. This front end is then run with the ported jKeystone, causing jKeystone to output a result that should correspond to the result produced by keystone. After this black-box test, aspects are woven through both systems to generate two sequence diagrams specific to the test case, and these are then compared as a white-box test. This process is described in more detail below.

4.3 Black box testing

Black box testing is a method of testing whereby only the input and output by the subject under test are analysed. The black box testing of the ported system was achieved quite readily using the test-suites described above.

We choose to use two outputs from keystone's static analysis as the test-case result criteria. The first of these is a summary of the tokens passed to the parser by the lexical analysis phase. keystone overcomes some of the ambiguity of ISO C++ by using a system known as *token decoration*, which, among other things, generates context-sensitive identifiers. Since the correct assignment of context is crucial to the parse, the output of the lexical analysis phase was used as test output. Since this output can be generated using a command-line switch in keystone, this still technically constitutes a black-box test.

The second output used as a criterion in the black-box tests is the main output of keystone, which consists of a detailed summary of the symbol table information, including a summary of all of the scopes and their members. This information is solely dependent on how the program is parsed and the output constitutes a relatively detailed summary of the input test-case. If the output of a test-case is identical for *both* criteria during black box testing then the test-case is considered to have passed. If the output is not identical, then the white box testing allows the location of the difference in execution traces to be readily identified for bug-fixing.

4.4 Dynamic Sequence Diagrams

Since keystone is also used as an API, it is important that the internal behaviour of the ported system correspond to the original. This cannot be readily verified by the end-to-end style of testing used in the black-box tests. White box testing involves the examination of the internals of the system under test.

Modern object-oriented features such as inheritance, polymorphism and dynamic binding mean that the effort in comprehending and reverse engineering a system from source code alone is high. Furthermore without access to an object's runtime type, it can become impossible to predict the flow of control through a system. The use of a UML sequence diagram can greatly aid in the comprehension of the behaviour of an object-oriented system [29].

Sequence diagrams are typically used in the design stage of a system, and depict the objects and interactions involved in a particular scenario of usage. However, given a system implementation, it is also possible to reverse engineer

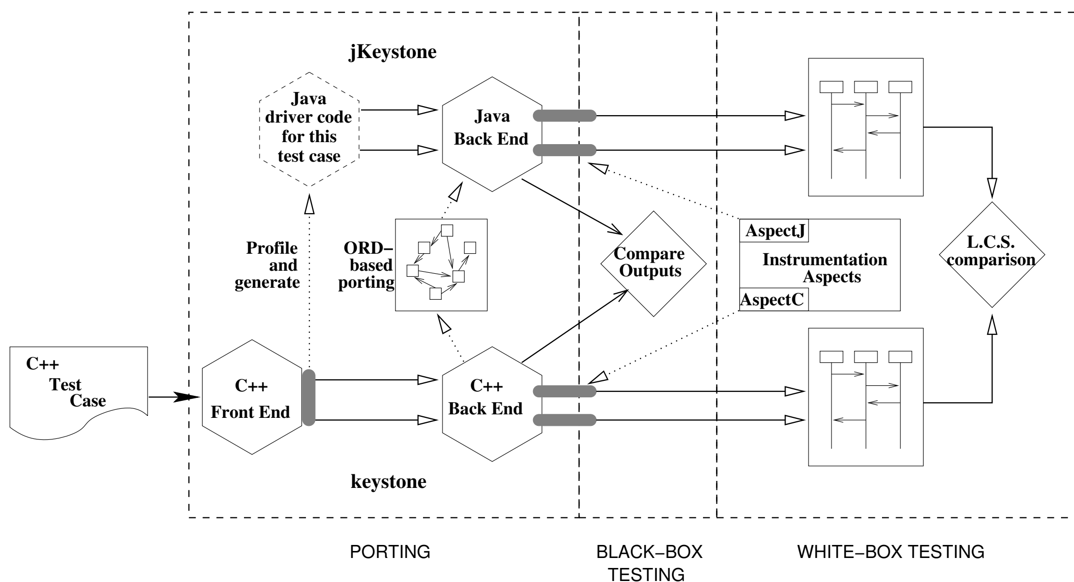


Figure 4: An overview of the porting process. *The lower half of the diagram represents the original* keystone *system in C++, and the upper half represents the ported* jKeystone *system in Java. The right part of the diagram shows the use of AOP to generate sequence diagrams, which are then compared using the LCS algorithm.*

sequence diagrams. These diagrams can be reverse engineered either statically [24] through an analysis of the program source or dynamically from program execution traces [19, 4] or more recently by utilising aspect-oriented programming [3].

4.5 Using Aspects to generate sequence diagrams

Upon completion of a black box test, each testcase was investigated further to ensure that the corresponding sequence diagrams for keystone and jKeystone were equivalent. To achieve this, it is necessary to generate a dynamic sequence diagram from both systems for each test case. This entailed instrumenting object creations as well as method calls and returns in both systems. As such, it is an instance of one of the canonical application areas for aspect oriented programming.

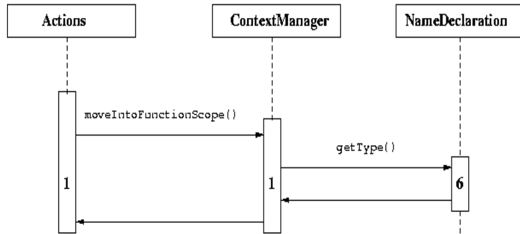
Through the use of AOP, it is possible to weave an aspect through a program that captures a trace of its execution. Following the standard AOP approach, a set of *pointcuts* are defined to trace method calls and returns, as well as constructor invocations. The *advice* executed at each of these pointcuts printed a relevant message to a logging file, maintaining the sequence of the method calls along with the current level of nesting.

Another feature of AOP is the use of *introductions*, which allow the aspect to make static changes to a class, such as introducing an extra attribute. This was used to assign a unique numeric identifier to each object being profiled, so that it could be explicitly identified. By using the same introduction in the C++ and Java versions of the aspect, we ensured that objects were being created in the same order, at the same point in the program, and could thus assign a specific owner-object to each method call.

In general, one advantage of AOP is that the aspect can be coded separately from the main system, and added and withdrawn as necessary. In the context of porting, AOP offers the possibility that the aspect can be written once, and then woven into both the original and ported system. As well as economy of effort, this also

Actions	1	function_definition_1()	1
ContextManager	1	moveIntoFunctionScope()	2
NameDeclaration	6	getType()	3

(a) A sample of the execution data logged for a method call.



(b) A UML sequence diagram derived from program trace.

Figure 5: An example of the execution data and corresponding UML sequence diagram for a method call. *For each call we record the owning class and object identifier, the function name and parameters, and the nesting level of the call.*

helps ensure that the behaviour of the resulting code is identical in each case, and any differences result from differences in behaviour of the systems themselves.

In practice, it was not possible to share identical aspects between both systems. The original keystone system was profiled using aspects woven by *AspectC* [27], which has almost reached maturity as a project since the preview release of version 1.0 of the program. The ported jKeystone system was profiled using aspects woven by *AspectJ* [14], a mature project that provides a compiler to weave aspects across Java source code. While the notation used by each aspect compiler is slightly different, the use of aspects did facilitate ensuring that the woven code performed similar actions in each case.

4.6 Comparison of Sequence Diagrams

The traces recorded by the aspect woven across keystone and jKeystone recorded the method currently executing, the owner-object and its unique identifier, and the current nesting level of the method. Figure 5(a) illustrates how our aspect captures the structure of a sequence diagram and Figure 5(b) shows its representation as a UML sequence diagram. Representing both traces textually allows them to be compared side by side to identify points at which program execution or object construction differ. However, as Table 1 illustrates, the size of the diagrams involved ensures that manual comparison of the diagrams is infeasible for each test-case. A method for automating the different diagrams for each trace is desirable. An overly simplistic approach would be to use a utility tool such as `diff` and report on any differences. However there are a number of important factors that mitigate against this approach.

The first of these is that the convention for the evaluation of arguments differs between Java and C++. According to the Java language specification [8], the arguments must be evaluated from left to right. This is not strictly the case in C++ which does not explicitly specify an order. However as a legacy carried over from C compilers, and especially on `x86` architectures, arguments are evaluated from right to left [13]. This small detail does not obviously affect the external behaviour of the systems but can change the generated sequence diagram if argument evaluatiob causes further method calls.

The second factor that must be accounted for is a potential difference in method names across both systems. Method names may not remain the same across both systems, *e.g.* the method name *clone* could not be ported to Java due a clash with the *clone* method belonging to `java.lang.Object`. A rudimentary comparison of two diagrams containing the same execution but with different method names would flag this as a difference. Another factor is a slight difference in coding idioms between C++ and Java. In the C++ version of keystone, an iteration over a list data structure results in two

calls, one a pointer to the begining of the list, another to the end. The Java version makes just one call via the new *for-each* loop present in Java version 1.5.

The final factor relates to the tools used for the tracing itself. *AspectC* is only moving towards maturity as a project and thus is incapable of parsing all C++ constructs fully. To overcome this is it possible to supply empty methods via macro definitions to *AspectC* to allow it to bypass difficult constructs. Thus for a very small percentage of the traces, the dummy code supplied to *AspectC* will cause its output to vary from that of *AspectJ*.

The issues outlined above mean that a comparison of the diagrams involves more than a simple use of `diff`. We have devised an automated approach to comparing the diagrams via a 5-step process. This approach involves:

1. The first step involves post-processing the Java version of the sequence diagram to fix the issue with the arguments being evaluated in opposite order to the C++ version.
2. The percentage similarity of the diagrams is calculated through the use of Hirschberg's algorithm for identifying the longest common subsequence (LCS) [11].
3. The inverse of the longest common subsequence is calculated from the output of Step 2.
4. The output from Step 3 is compared to all known differences that can occur through different method names or through a limitation of *AspectC*.
5. If there are any remaining elements in the inverse that cannot be accounted for then the test-case had failed the white-box test and a bug fix must take place.

Hirschberg's LCS algorithm compares two sequences of characters and computes the longest common subsequence of (not necessarily contiguous) characters that they have in common. Using a full line of the sequence diagram as seen in Figure 5(a) for the characters, applying the LCS algorithm to a list of method calls, and taking the inverse of the LCS with the jKeystone sequence diagram, yields those method calls in jKeystone that did not occur in the original keystone. These are evaluated, a patch is applied, and the process is iterated until the sequences are identical.

	Objects created		**Methods executed**	
	jKeystone	keystone	jKeystone	keystone
Min	3	3	8	8
1st Qu.	15	15	638	640
Median	23	23	1087	1091
3rd Qu.	34	34	1792	1800
Mean	31.50	31.50	2152	2152
Max	1150	1150	216141	216350

Table 1: A summary of sequence diagram size for the 1761 test-cases. *The results are partitioned into objects created and methods executed during the running of the test-cases. As well as the minimum and maximum encountered, the first, second and third quartiles alongside the mean are also given.*

5 Results

In this section we outline the results of our test-driven porting strategy. We present a summary of the sequence diagram size and the overhead involved in their generation. We also present a classification of the types of bugs we have discovered through our testing.

5.1 Sequence diagrams

In order to demonstrate the scale of the activity involved in generating sequence diagrams, Table 1 gives a statistical summary of the UML sequence diagrams generated by both keystone and jKeystone. The results are partitioned according to the number of distinct objects present and the number of method calls in a sequence diagram. Within each partition, we further sub-divide the results between jKeystone and keystone. The first row in Table 1 gives the minimum number of objects created and methods executed in any single sequence diagram. We present the same results for the first, second and third quartile. The average number of objects created and methods called is given in the fifth row while the maximum values are provided in the sixth row.

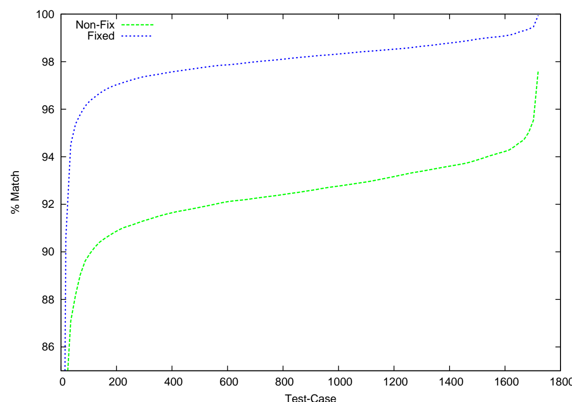


Figure 6: An example of comparing sequence diagrams across all of the test-cases. *The test cases are listed along the x-axis, and y-axis represents the percentage match between the sequence diagrams generated from* jKeystone *and* keystone. *The lower and upper lines plot the percentage match for each test case before and after the fix for method argument evaluation is applied.*

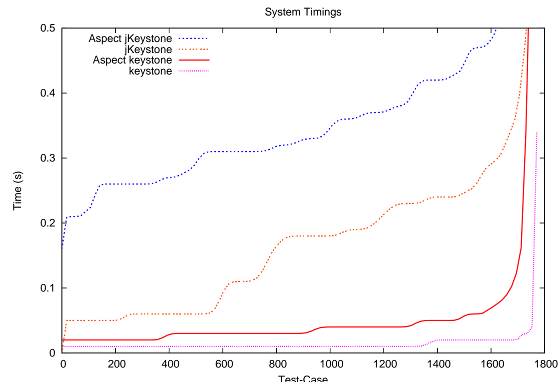


Figure 7: The timing results for each test case. keystone *and the aspected version of* keystone *are the two bottom plots whilst* jKeystone *and the aspected* jKeystone *form the top two respectively. Test-cases with timings greater than 500ms have been removed for clarity.*

As well as demonstrating the infeasibility of manually comparing the sequence diagrams, the descriptive statistics in Table 1 provide a broad indication that the two systems are performing similarly. Object creation is a crucial operation of any object-oriented system and as can be seen from the average column, the number of object creations in both systems is identical across the 1761 test-cases. Furthermore the number of method calls is, on average, the same across both systems. However this number can fluctuate between test-cases, as outlined in Section 4.6 above, and further analysis on a case-by-case basis is crucial to ensuring correctness.

A more precise quantification of the discrepancies between keystone and jKeystone is given by applying the LCS algorithm and examining the differences. These differences in the sequence diagram contents, exposed by the LCS algorithm and expressed as a percentage of the size of the keystone sequence diagram, served as a metric of progress of the white-box testing. That is, the LCS algorithm was used not only to isolating bugs during testing and to ensure ultimate 100% identity between generated sequence diagrams, but also as a measure of the rate of progress of the white-box testing phase.

An example of using the LCS to quantify similarity between keystone and jKeystone at an early stage of white-box testing is given in Figure 6. This figure plots the percentage similarity between the two sequence diagrams before and after the fix for the method argument ordering is applied. The lower line plotted in Figure 6 shows the jKeystone sequence diagrams for approximately 1700 of test-cases exhibiting similarity of over 90% with keystone before any fix is applied. The upper line plotted in Figure 6 shows this increasing to over 97% for most of the test-cases once the fix is applied. Ultimately bringing this figure to 100% for all test cases allows us to determine with certainty when the white-box testing phase is complete.

5.2 Instrumentation overhead

In order for a testing strategy to be used it must be practicable; in particular, it must not impose an unreasonable burden on the tester. While the strategy of using UML sequence diagrams is based on an existing test suite, it is possible that the overhead of generating the diagrams would impose a significant overhead on the testing process. One immediate advantage of using AOP is that we can turn sequence diagram generation on or off very easily.

To examine the overhead, we tracked the timing of generating sequence diagrams for both keystone and jKeystone, and the results are summarised in Figure 7. The timings were performed on a *Dell* Optiplex GX280 PC, with a 3.06Ghz Intel processor, 1Gb DDR RAM running the Fedora Core 4 distribution of GNU/Linux.

The graph in Figure 7 shows four sets of timings, one each for keystone with and without sequence diagram generation, and one each for jKeystone with and without sequence diagram generation. Each point on the graph represents a single test case, and the test-cases have been arranged in increasing order of size along the horizontal axis.

The first point to note in Figure 7 is the C++ / Java divide in the timings. The C++ based keystone gives an almost constant 10 milliseconds for almost 1400 of the test-cases whereas the aspected keystone takes twice the time to generate the sequence diagrams. The timings for jKeystone range between 300 and 400 milliseconds for approximately the first 1400 test-cases. When sequence diagram generation is turned on, the timings rise by approximately 50% across the test-cases.

This divide between the systems is not unexpected. Many of the test cases are quite small, and the overhead of JVM startup imposes a significant penalty on jKeystone. It is positive to note, however, that the aspects do not place an unfeasible bound on the time taken to analyse a given test-case in either keystone or jKeystone.

5.3 Bug classification

Table 2 gives a breakdown of the bugs that were discovered using the sequence diagrams during the system tests. In total 30 unique bugs were discovered during the system testing phase. The bugs are classified according to the IEEE classification for software anomalies [25], the number of each instance of anomaly and the specific occurrence of the anomaly within jKeystone. The table is partitioned along the following lines: bugs due to misunderstanding of the original code and bugs introduced during porting. Rows 1-3 highlight the bugs introduced through misunderstandings whilst 4-10 are the bugs inadvertently entered during porting.

Anomaly	Freq.	Actual Occurrence
IV310	3	Assertions incorrect
IV315	4	Code ported incorrectly
IV316	9	Guard clause inadequate
IV321	1	Guard clause too strong
IV317	3	Wrong variable checked
IV321.4	2	Scope of statement incorrect
IV341	3	Object initialised incorrectly
IV342	2	Accessed incorrect data
IV342.1	1	Return type hard-coded
IV342.4	2	Data accessed out of bounds

Table 2: A classification of the bugs identified during out system tests. *The IEEE anomaly index is given along with the frequency of the bug and the actual manifestation of the bug.*

The bugs featured in 1-3 of Table 2 are the result of a shortfall in the comprehension of specific areas of keystone. The largest number of bugs within this category are from **IV 316**: "*missing condition test*". This is due to the fact that a pointer can be dynamically cast within an if statement in C++. This cast check was missing initially from the Java version. The bugs featured in 4-10 of Table 2 are bugs that entered jKeystone through human error. These bugs are spread over a number of anomaly classifications and can be considered to be "one-off" bugs. That is, the discovery of a bug did not unearth a plethora of similar bugs. Thus the use of the dynamic UML sequence diagrams was essential in isolating the difference in method calls across the two systems and discovering the approximate location of the bug within jKeystone.

6 Conclusion and Future Work

In this paper we have tested the practical use of porting a medium sized C++ system to Java through the use of an Object Relation Diagram. The construction of an ORD via the algorithm outlined in Section 3.2 within the eXtreme Porting strategy allows the porting to take place in a manner that minimises the need for stubs during unit testing.

In addition to ensuring that the structure

of the original system remains preserved under porting, it is possible to create a sequence diagram for the purposes of comparison. Thus, a crucial phase of our testing is the ability to compare the two reverse engineered sequence diagrams side-by-side to prove that the behaviour is preserved across both systems. It is worth noting that more complex approaches for comparing sequence diagrams exist (e.g. reconciliation [26]) and could be used with our approach, but for our purposes the quick discovery of differences in the sequence diagrams were used to locate, isolate and fix anomalies within the ported code.

At present, the use of record and playback restricts us to using *positive* test-cases. In our future work, we plan to investigate the possible contribution of *negative* test-cases to ensure correctness. The comparison of objects within the sequence diagrams was at a coarse level in our study and although this worked for us, other approaches may need a finer level of granularity. Finally we also plan to use a generalised LR parser that is under development as the parser front-end for jKeystone.

7 Biography

Mark Hennessy is a PhD. candidate in the Department of Computer Science at NUI Maynooth. His principal research focus is on the test-driven development of grammar-based software, with an emphasis on ISO C++.
Dr. James F. Power is a lecturer in the Department of Computer Science at NUI Maynooth. His research interests include program comprehension, reverse engineering and software visualisation.

References

[1] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.

[2] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, July 1998.

[3] L. Briand, Y. Labiche, and J. Leduc. Tracing distributed systems executions using AspectJ. In *21st International Conference on Software Maintenance*, pages 81–90, Budapest, September 2005.

[4] L. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In *10th Working Conference on Reverse Engineering*, pages 57–66, Victoria, BC, Canada, November 2003.

[5] L. Briand, Y. Labiche, and Y. Wang. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. In *12th International Symposium on Software Reliability Engineering*, pages 287–296, Hong Kong, November 2001.

[6] T. H. Gibbs, B. A. Malloy, and J. F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software: Practice and Experience*, 33(1):19–39, January 2003.

[7] T. H. Gibbs, B. A. Malloy, and J. F. Power. Progression toward conformance of C++ language compilers. *Dr. Dobbs Journal*, 28(11):54–60, September 2003.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2005. Third Edition.

[9] M. Hennessy and J. F. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *20th International Conference on Automated Software Engineering*, pages 104–113, Long Beach, CA, USA, November 2005.

[10] M. Hennessy and J. F. Power. Generation strategies for test-suites of grammar-based software. Technical Report NUIM-CS-TR-2005-02, Department of Computer Science, National Univesity of Ireland, Maynooth, April 13 2005.

[11] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.

[12] D. Hoffman and K. Yoo. Blowtorch: a framework for firewall test automation. In *20th International Conference on Automated Software Engineering*, pages 96–103, Long Beach, CA, USA, November 2005.

[13] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. American National Standards Institute, first edition, September 1998.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[15] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. A test strategy for object-oriented systems. In *Computer Software and Applications Conference*, pages 239 – 244, Dallas TX., August 1995.

[16] B. A. Malloy, P. J. Clarke, and E. L. Lloyd. A parameterized cost model to order classes for class-based testing of C++ applications. In *14th International Symposium on Software Reliability Engineering*, pages 353–364, Denver, CO., November 2003.

[17] B. A. Malloy, S. A. Linde, E. B. Duffy, and J. F. Power. Testing C++ compilers for ISO language conformance. *Dr. Dobbs Journal*, 27(6):71–78, June 2002.

[18] B. A. Malloy and J. F. Power. An interpretation of Purdom's algorithm for automatic generation of test cases. In *1st Annual International Conference on Computer and Information Science*, Orlando, FL., October 2001.

[19] B. A. Malloy and J. F. Power. Exploiting UML dynamic object modeling for the visualization of C++ programs. In *ACM Symposium on Software Visualisation*, pages 105 – 114, St. Louis, MO, USA, May 2005.

[20] R. C. Martin. *Agile Software Development*. Prentice Hall, 2003.

[21] A. Memon. GUI testing: Pitfalls and process. *IEEE Computer*, 35(8):87–88, August 2002.

[22] A. Milanova, A. Rountev, and B. Ryder. Constructing precise object relation diagrams. In *International Conference on Software Maintenance*, pages 586–595, Montreal, Canada, September 2002.

[23] M. Roper. *Software Testing*. McGraw-Hill, 1994.

[24] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *27th International Conference on Software Engineering*, pages 254–263, St. Louis, MO, USA, May 2005.

[25] Software Engineering Standards Committee of the IEEE. *IEEE Standard Classification for Software Anomolies*. IEEE Standards Board, 1993.

[26] G. Spanoudakis and H. Kim. Supporting the reconciliation of models of object behaviour. *Software and Systems Modeling*, 3(4):273–293, December 2004.

[27] O. Spinczyk, D. Lohmann, and M. Urban. AspectC++: an AOP extension for C++. *Software Developer's Journal*, pages 68–76, May 2005.

[28] K.-C. Tai and F. Daniels. Test order for inter-class integration testing of object-oriented software. In *Computer Software and Applications Conference*, pages 602–607, Washington, DC, USA, August 1997.

[29] The Object Management Group. The Unified Modelling Language Version 1.5 OMG. Formal/2003-03-01.

[30] P. Varma, A. Anand, D. P. Pazel, and B. R. Tibbitts. Nextgen extreme porting: structured by automation. In *ACM Symposium on Applied Computing*, pages 1511–1517, Santa Fe, New Mexico, March 2005.