Utilize Syntax Tree Transformations as a C/C++ Test Seam

GÁBOR MÁRTON and ZOLTÁN PORKOLÁB, Eötvös Loránd University

In C/C++, test code often influences the source code of the unit we want to test. During the test development process we often have to introduce new interfaces to replace existing dependencies, e.g. a supplementary setter or constructor function has to be added to a class for dependency injection. In many cases, extra template parameters are used for the same purpose. These solutions may have serious detrimental effects on the code structure and sometimes on the run-time performance as well. We can use non-intrusive tests (tests which does not require any modification in the production code) to avoid these disadvantages.

Also, in legacy code bases often there are few or no unit tests. Refactoring such code in order to provide tests is almost impossible because we cannot verify correctness without having unit tests; hence it is a vicious circle. We can break the circle with non-intrusive tests, i.e. without actually modifying the production code.

The different non-intrusive testing methods have different weaknesses. In this paper we introduce our new non-intrusive testing approach which complements the existing techniques. Our solution transforms certain parts of the original abstract syntax tree of the production code for the purpose of testing.

1. INTRODUCTION

Testing is essential in modern software development [Bertolino and Marchetti 2004; Khan and Khan 2014; Fowler 2005; Majchrzak 2012] to improve the quality of a system and reduce the cost of maintenance. There are different layers of testing from unit tests to stability, functional and integration tests. In this paper we focus on unit testing, which is the most language-specific method. Unit testing an object with unwanted dependencies may be very problematic, because one dependency may represent a database or a network connection, whose behaviour can be hard or expensive to simulate. Thus, during unit testing of a system eliminating unwanted dependencies is necessary. In object-oriented programming languages, this dependency replacement often requires the modification of the original public interface of the unit under test. For instance, new setter or constructor functions have to be added to a class or we have to promote a concrete type to a class template. Nevertheless, there are cases when these new functions are not intended to be used in production code or using a template is not desired because of the existing program structure. Moreover, in C++, source code modification for testing could result in performance degradation, e.g. introducing a new runtime interface and virtual functions just because of testing might worsen the performance of the production code. Also ,in legacy code bases often there are few or no unit tests. Refactoring such code in order to provide tests is almost impossible because we cannot verify correctness without having unit tests; hence it is a vicious circle.

Author's addresses:
Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages and Compilers, Pázmány Péter sétány 1/C, Budapest, Hungary, H-1177, `martongabesz@gmail.com`, `gsd@elte.hu`

This work is supported by the European Union, co-financed by the European Social Fund
(EFOP-3.6.3-VEKOP-16-2017-00002).

In: Z. Budimac (ed.): Proceedings of the SQAMIA 2018: 7th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia, 27–30.8.2018. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

We can break the circle with non-intrusive tests, i.e. without actually modifying the production code [Feathers 2004; Rüegg and Sommerlad 2012]. We refer to all those testing approaches which require source code modification as *intrusive* testing and those which does not as *non-intrusive* testing.

An *abstract syntax tree* (AST) is a data structure which represents the hierarchical syntactic structure of the source program [Aho et al. 1986]. They are widely used in compilers as an intermediate representation of the program through several stages that the compiler requires. During certain compilation stages the abstract syntax tree is being transformed, e.g. in case of C++ template instantiation existing nodes are being modified and new nodes are added to the tree.

In this paper, we investigate a new, non-intrusive, syntax tree transformation based testing approach. This paper is organized as follows. In Section 2 we describe existing unit testing methods to replace dependencies. We show how we can replace dependent functions or types with syntax tree transformations in Section 3. In Section 4, we outline future work and current limitations. Our paper concludes in Section 5.

2. TEST SEAMS IN C/C++

A *seam* is an abstract concept introduced by Feathers to identify points where we can break dependencies [Feathers 2004]. The goal is to have a place where we can alter the behaviour of a program without modifying it in that place; this is important because editing the source code is often not an option [Rüegg and Sommerlad 2012]. Feathers, Rüegg and Sommerlad define four different kinds of seams for C++ [Feathers 2004; Rüegg and Sommerlad 2012; mockator.com 2006]:

(1) Link seam: Change the definition of a function via some linker specific setup.
(2) Preprocessor seam: With the help of the preprocessor, redefine function names to use an alternative implementation.
(3) Object seam: Based on inheritance to inject a subclass with an alternative implementation.
(4) Compile seam: Inject dependencies at compile-time through template parameters.

The *enabling point* of a seam is the place where we can make the decision to use one behaviour or another. Different seams have different enabling points. For example, replacing the constructor argument for the implementation of an interface with a mock implementation when a unit test is set up is an object seam with the constructor as an enabling point.

Link and preprocessor seams can be used to write non-intrusive tests. However, object and compile seams may be used for such purpose only if the unit under test already has the proper architecture. For instance, in case of object seams the unit must have a constructor (or setter) function to setup a different implementation for the dependency. In case of compile seams, the unit must be a template and it must have a template parameter via which we can mock the dependency. Often, these architectural requirements are not satisfied, therefore the use of object and compile seams demand that we intrusively change the source code of the unit.

2.1 Link Seams

We can use a link seam e.g. to replace the implementation of a free function or a member function as presented in Figure 1. On the one hand, when we need to test the `bar()` function then we should link the test executable to the `MockA.o` object file. On the other hand, we should link the production code with `A.o`. Link-time dependency replacement is not possible if the dependency is defined in a static library or in the same translation unit where the SUT is defined. It is also not feasible to use link seams if the dependency is implemented as an inline function [Rüegg and Sommerlad 2012]. This makes the use of this seam cumbersome or practically impossible when the dependant unit is a template or when

```
// A.hpp
void foo();
```

```
// A.cpp
void foo() { ... };
```

```
// MockA.cpp
void foo() { ... };
```

```
// B.cpp
#include "A.hpp"
void bar() { foo(); ... }
```

Fig. 1. An example of a link seam

the dependency is a template. The enabling point for a link seam is always outside of the program text. This makes the use of link seams quite difficult to identify. On top of all, link-time substitution requires strong support from the build system we are using. Thus, we might have to specialize the building of the tests for each and every unit. This does not scale well and can be really demanding regarding to maintenance.

2.2 Preprocessor Seams

Preprocessor seams can be applied to replace the invocation of a global function to an invocation of a test double [Mihalicza et al. 2011]. Let us consider the code snippet in Figure 2. We can replace the standard `malloc()` and `free()` functions with our own implementation. One example usage may be to collect statistics or do sanity checks in `my_malloc` and `my_free` functions. These seams can be applied conveniently in C, but not in C++. As soon as we use namespaces, the preprocessor might generate code which cannot be compiled because of the ambiguous use of names. Hazardous side effects of macros are also well known.

```
void *my_malloc(size_t size) {
  //...
  return malloc(size);
}
void my_free(void *p) {
  //...
  return free(p);
}
```

```
#define free my_free
#define malloc my_malloc

void unitUnderTest() {
  int *array = (int *)malloc(4 * sizeof(int));
  // do something with array
  free(array);
}
```

Fig. 2. An example of a preprocessor seam

In our previous work we introduced a compiler instrumentation based test seam, which does not have the mentioned drawbacks of link and preprocessor seams [Marton and Porkolab 2017; Márton 2017]. However, that technique is still not applicable for all cases, for example we can replace simple function dependencies only but we cannot replace types.

3. THE AST TRANSFORMATION SEAM

Existing test seams for C++ have some disadvantages that prevents us from using them or make us reluctant to use them. Link seams do not work with inline functions and require patching the build system. Preprocessor seams are problematic with classes and namespaces. Object seams and compile seams are intrusive and often demand that we widen the public interface. Therefore, we seek for a new seam which does not have the above-mentioned disadvantages and makes it possible to write non-intrusive tests and to replace dependent types with test double types.

3.1 AST Transformations

Abstract syntax tree transformations are used for several purposes. In the groovy programming language [Koenig et al. 2007] it is used as a form of compile-time metaprogramming; it allows the developers to hook into the compilation process and to modify the AST before bytecode generation.

The LLVM/Clang compiler infrastructure [Lattner 2008] makes it possible to write certain source to source transformations which modify the existing AST of a source file and produces the modified source code as an output. This may be used for instance, to annotate the source code with additional statements that create statistics about memory allocations.

CodeBoost is a source-to-source transformation tool for domain-specific optimisation of C++ programs [Bagge et al. 2003; Bagge and Haveraaen 2003]. CodeBoost performs parsing, semantic analysis and pretty-printing, and transformations can be implemented either in the Stratego program transformation language, or as user-defined rewrite rules embedded within the C++ program.

Constant propagation, a well known data-flow optimization problem, can be implemented on abstract syntax trees in Stratego, a rewriting system extended with programmable rewriting strategies for the control over the application of rules and dynamic rewrite rules for the propagation of information [Olmos and Visser 2002].

Prolog can be retrofitted with concrete syntax and a seamless interaction of concrete syntax fragments with an existing ”legacy” meta-programming system (based on abstract syntax) can be achieved [Fischer and Visser 2003]. This can result in a considerable reduction of the code size and an improved readability of the Prolog source code.

3.2 The AST Import Algorithm

Traditionally, C/C++ compilers process one *translation unit* (TU) at a time. However, there are use cases when we have to access information from several (or all) translation units of a program. For example, in case of link time optimization, or in case of cross translation unit static analysis. The LLVM/Clang cross translation unit static analysis [Horvath 2017] executes the same single translation unit analysis algorithms but on an AST which is synthesized by merging several ASTs of the individual translation units. This merged AST is created by first parsing one TU and then importing the ASTs of the other TUs one-by-one into the first.

Not considering templates, two AST nodes are *structurally equivalent* if they are

- builtin types and refer to the same type, e.g. `int` and `int` are structurally equivalent,
- function types and all their parameters have structurally equivalent types,
- record types and all their fields in order of their definition have the same identifier names and structurally equivalent types,
- variable or function declarations and they have the same identifier name and their types are structurally equivalent.

Algorithm 1 describes the procedure of importing an AST. The algorithm has to ensure that the structurally equivalent nodes in the different translation units are not getting duplicated in the merged AST. E.g. if we include the definition of the vector template (`#include <vector>`) in two translation units, then their merged AST should have only one node which represents the template. Also, we have to discover *one definition rule* (ODR) violations. For instance if there is a class definition with the same name in both translation units, but one of the definition contains different number of fields. We refer to the translation unit into which we import as the *to* TU and the one which we import from as the *from* TU.

3.3 Reusing the AST Import Algorithm for Testing

As our contribution we extended the AST importer mechanism. With this extension we can transmogrify the original AST of the production code to an AST which contains the nodes of test doubles instead of the nodes of the original dependencies. The key of our approach is that we import everything into the test double's context. When the production code is being merged and when we reach the import

ALGORITHM 1: AST Import Algorithm

```
for each top level $Decl$ in the $From$ TU do
    $FoundDeclsList \leftarrow$ lookup all declarations in the $To$ TU with the same name of $Decl$
    for each $FoundDecl$ in $FoundDeclsList$ do
        if $FoundDecl$ is structurally equivalent with $Decl$ then
            $ToDecl \leftarrow FoundDecl$
            mark $Decl$ as imported
        end
        else if $Decl$ is a function then overloaded function
            $ToDecl \leftarrow$ create a new AST node in $To$ TU
            import dependent declarations and types of $ToDecl$
        end
        else
            report ODR violation
        end
    end
    if $FoundDeclsList$ is empty then
        $ToDecl \leftarrow$ create a new AST node in $To$ TU
        import dependent declarations and types of $ToDecl$
    end
end
```

of a dependency then the lookup finds the definition of the test double and that definition is getting used in the rest of the imported AST. We introduced a new attribute (`[[test_double]]`) which modifies the behaviour of the structural equivalence check. During the check of two declarations/types, if this attribute is present in the declaration/type of the ”to” context then we consider the two entity as equivalent. With the help of this new attribute and the modified equivalence check we are able to change the import algorithm to use the test double in the production code. We created a prototype implementation based on the LLVM compiler infrastructure which is publicly available at [Márton 2018].

Figure 3 demonstrates how our new method can be used to replace a simple function in the AST. The `foo()` function is defined in the `foo.c` translation unit and it calls the `bar()` function, which is defined in the very same TU. The `fake_bar.c` file contains the definition for the test double which we want to use as the replacement function in the test. The test double has the same name as the original function we want to replace. Normally, having two definitions would cause ODR violation during the import procedure, but this time the test double has the special attribute attached to its definition. In `test.c` we have the actual test code, which exercises the production code (`foo()`) and formulates expectations on that. If `foo()` calls the test double `bar()` then the return value will be `13` and then `main()` will return with success. First we have to create the serialized AST files for each source file (`-emit-pch`). Then we import the AST of the production code (`foo.ast`) and the test code (`test.ast`) into the AST of the test double (`fake_bar.ast`). We achieve this by providing the AST files in the proper order and with the `-ast-merge` command line option of the compiler. Once we have the merged AST then we emit an object file from that (`-emit-obj`) and during this process all C/C++ semantic checks are executed. Thus, if the modified AST is semantically incorrect then we will be notified.

In Figure 4 we show how we can replace a record type. This time we have a class as a dependency (`Bar`) defined in `foo.h`. The test double in `fake_bar.h` simply replaces the only `f()` member function, which returns a fake value. The rest of the mechanism is quite similar as before, the only difference is that this time we have to generate AST files from headers.

The previous example in Figure 4 presented that we can easily replace types if the substitute type has the same functions defined as the original one. Figure 5 demonstrates we can even extend the

```
// foo.c
int bar() { return 1; }
int foo() {
  return bar();
}
```

```
// fake_bar.c
[[test_double]]
int bar() {
  return 13;
}
```

```
// test.c
int foo();
int main() {
  return
      foo() == 13 ? 0 : 1;
}
```

```
# Create the ASTs
clang -cc1 -emit-pch -o foo.ast foo.c
clang -cc1 -emit-pch -o fake_bar.ast fake_bar.c
clang -cc1 -emit-pch -o test.ast test.c

# Merge the ASTs and emit an object
clang -cc1 -ast-merge fake_bar.ast -ast-merge foo.ast -ast-merge test.ast /dev/null \
  -emit-obj -o merged.o
# Link
clang -o output merged.o

# Run the test
./output
```

Fig. 3. Replacing a simple function in the AST

```
// foo.h
class Bar {
  int a;
public:
  int f() { return 1; }
};

class Foo {
  Bar bar;
public:
  int ff() {
    return bar.f();
  }
};
```

```
// fake_bar.h
class [[test_double]] Bar {
public:
  int f() {
    return 13;
  }
};
```

```
// test.c
#include "foo.h"

int main() {
  return Foo().ff() == 13
            ? 0
            : 1;
}
```

Fig. 4. Replacing a record type in the AST

replaced type with additional functionality, which may be very useful if we want to create mock test doubles and not just simple stub objects. In the test (`test.c`) we would like to use a test double which can be setup to return with a given value in its member function (`f()`). To achieve this we provide a new header file (`mock_bar_modifiers_fwd.h`) which contains prototype definitions for such functions which can modify the dependency via a pointer. Since we use a pointer we do not have to see the whole definition of the `Bar` class a forward declaration is sufficient. With this auxiliary header file we can create the AST dump for the test file. The definition of the setter function(s) are placed in the test double file (`mock_bar.h`) together with the definition of the mock type. The rest of the mechanism is similar to the previous examples, we have to merge the AST files of the production and test code into the AST of the test double. There is no need to create a separate AST file for the auxiliary header.

3.4 Evaluation

Compared to link seams, our new seam makes it possible to replace functions even if they are inlined or statically linked. Besides, we can replace class definitions and virtually anything because we do the replacement on the AST level. Similarly to link seams, our method requires additional help from the

```
// foo.h
class Bar {
  int a;
public:
  int f() { return 1; }
};

class Foo {
public:
  Bar bar;
  int ff() { return bar.f(); }
};
```

```
// mock_bar_modifiers_fwd.h
struct Bar;
void set_f_return_value(Bar* bar, int value);
```

```
// mock_bar.h
#include "mock_bar_modifiers_fwd.h"

struct [[test_double]] Bar {
  int f_return_value = 0;
  int f() {
    return f_return_value;
  }
};

void set_f_return_value(Bar* bar, int value) {
  bar->f_return_value = value;
}
```

```
// test.c
#include "foo.h"
#include "mock_bar_modifiers_fwd.h"

int main() {
  Foo foo;
  set_f_return_value(&foo.bar, 13);
  return foo.ff() == 13 ? 0 : 1;
}
```

Fig. 5. Changing the AST to use a mock test double

build system since we have to create the AST files separately and we have to merge them manually. Also the enabling point of our seam is in the build scripts, as in the case of the link seams.

Similarly to the preprocessor seams, with our approach we can replace not just functions but types or anything which has a name. However, our approach does the replacement on the AST level, while the preprocessor seam does that on the token level.

4. LIMITATIONS AND FUTURE WORK

It is a common limitation with the three seams (link, preprocessor and astimporter) that we can replace only those dependencies which have an identifier name. E.g., it is not possible to replace a C++11 lambda function nested in a call of an algorithm from the standard template library:

```
std::vector<int> nums{1, 2, 3, 4, 5};
std::for_each(nums.begin(), nums.end(), [](int &n){ n++; });
```

As of this writing, the implementation of AST importer algorithm in LLVM/Clang is quite immature. Especially, the import of C++ templates and C++11 expressions are not well supported. Thus, it is an important future work to improve this realization because that would open up the possibility to be able to experiment with our new technique on real C++ projects.

5. CONCLUSION

Unit testing plays a crucial role in modern software development. The existing testing methods either require intrusive changes in the original source of the production code or they have some sort of disadvantages which makes them hard to use. In this paper we presented a new unit testing technique which is based on transforming the AST of the production code in a way that dependent functions or types are replaced with test doubles. This method has clear advantages compared to previous solutions because we can replace types as well, not just simple functions. Also, the technique exploits a compiler aided syntax tree modification which provides much safer transformation than we can get via tokeniser based translations with the preprocessor. Similarly to linker based solutions, our method requires additional help from the build system. Our new approach provides an alternative way to create non-intrusive tests, thus making it easier to verify legacy software or to write tests without bringing in unwanted structural changes in the production code. A prototype implementation based on the LLVM/Clang compiler infrastructure is publicly available at [Márton 2018].

REFERENCES

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, Principles, Techniques. *Addison Wesley* 7, 8 (1986), 9.

Otto Skrove Bagge and Magne Haveraaen. 2003. Domain-Specific Optimisation with User-Defined Rules in CodeBoost. In *Proceedings of the 4th International Workshop on Rule-Based Programming (RULE'03) (Electronic Notes in Theoretical Computer Science)*, Jean-Louis Giavitto and Pierre-Etienne Moreau (Eds.), Vol. 86/2. Elsevier, Valencia, Spain.

Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. 2003. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*. IEEE, 65–74.

Antonia Bertolino and Eda Marchetti. 2004. 1 A Brief Essay on Software Testing. (2004).

Michael Feathers. 2004. *Working Effectively with Legacy Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Bernd Fischer and Eelco Visser. 2003. Adding concrete syntax to a prolog-based program synthesis system. In *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, 56–58.

Martin Fowler. 2005. The New Methodology. (2005). http://www.martinfowler.com/articles/newMethodology.html

Gabor Horvath. 2017. Cross Translation Unit Analysis in Clang Static Analyzer. (2017). https://www.youtube.com/watch?v=7AWgaqvFsgs EuroLLVM.

Mohd. Ehmer Khan and Farmeena Khan. 2014. Importance of Software Testing in Software Development Life Cycle. (2014).

Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. 2007. *Groovy in action*. Manning Publications Co.

Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*. 1–2.

Tim A. Majchrzak. 2012. *Improving Software Testing: Technical and Organizational Developments*. Springer Publishing Company, Incorporated.

Gábor Márton. 2017. Instrumentation for testing. (2017). https://github.com/martong/finstrument_mock

Gábor Márton. 2018. Modified Clang ASTImporter for testing. (2018). https://github.com/martong/clang/tree/mock_with_astimporter

Gabor Marton and Zoltan Porkolab. 2017. Unit Testing in C++ with Compiler Instrumentation and Friends. *Acta Cybernetica* 23, 2 (2017), 659–686.

József Mihalicza, Zoltán Porkoláb, and Abel Gabor. 2011. Type-preserving heap profiler for C++. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*. IEEE Computer Society, 457–466. `DOI`:http://dx.doi.org/10.1109/ICSM.2011.6080813

mockator.com. 2006. An Eclipse CDT plug-in for C++ Seams and Mock Objects. (2006). http://mockator.com/

Karina Olmos and Eelco Visser. 2002. Strategies for Source-to-Source Constant Propagation. In *Workshop on Reduction Strategies (WRS'02) (Electronic Notes in Theoretical Computer Science)*, B. Gramlich and S. Lucas (Eds.), Vol. 70. Elsevier Science Publishers, Copenhagen, Denmark, 20. http://www.elsevier.nl/locate/entcs/volume70.html.

Michael Rüegg and Peter Sommerlad. 2012. Refactoring Towards Seams in C++. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST '12)*. IEEE Press, Piscataway, NJ, USA, 117–123. http://dl.acm.org/citation.cfm?id=2663608.2663632