# C++ Program Slicing with TRANSFORMERS

**Florian Quèze**

Technical Report $n^o$0825, May 2008
revision 1788

TRANSFORMERS is a C++ manipulation framework built on Stratego/XT. Program Slicing is an important field of program transformation. We will explain what Program Slicing is, give a quick overview of various aspects of Program Slicing and show how Transformers can be turned into a C++ Program Slicing tool.

TRANSFORMERS est un emsemble d'outils basés sur Stratego/XT permettant la manipulation de programmes C++. Le découpage de programmes est un domaine important de la transformation de programmes. Nous allons expliquer ce qu'est le découpage de programmes, donner un aperçu rapide de ses différents aspects et montrer comment Transformers pourrait être utilisé comme un outil permettant le découpage de programmes.

**Keywords**
Transformers, program slicing, C++

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
florian.queze@lrde.epita.fr – http://transformers.lrde.epita.fr

## Copying this document

# Contents

# Chapter 1

# Introduction

Program Slicing is an important field of program transformation with various aspects. Researches have been going on for a long time on this subject and there are now several concrete applications.

TRANSFORMERS is a C++ manipulation framework built on Stratego/XT. Even if its development is not really finished yet, we believe that it is time to start working on some examples of transformations that will illustrate its power. Hence the idea of doing C++ Program Slicing with Transformers.

In chapter 2 we will explain what Program Slicing is and give a quick overview of various kinds of Program Slicing tools. Then in chapter 3, we will detail a few common applications of Program Slicing. Finally, in chapter chapter 4, we will show how Transformers can be turned into a C++ Program Slicing tool.

**Acknowledgments** Thanks Maxime Van-Noppen for proofreading this report.

# Chapter 2

# Program Slicing

Program Slicing was first described in 1984 by Weiser (Weiser, 1984). Basically, slicing a program is reducing it to the minimal amount of code while preserving some interesting properties of the original program. The usual definition for a program slicing algorithm is that with a given program point p and a given set of variables V, it should produce the minimal program that will result in the variables of V having the same value at the program point p. The pair <p, V> is called slicing criterion.

Most program slicing problems are undecidable (Weiser, 1984, 1979). The program slicing tools try to delete as many pieces of code as possible, but when it is not possible to decide whether some chunk of code is really unused, the algorithms have to be conservative, to ensure they produce valid programs as output and they do not change the interesting properties of the original program.

Various kinds of program slicing techniques appeared over the time to fit different needs.

## 2.1 Static and dynamic program slicing

The program slicing tools can be separated in two big families: static slicing and dynamic slicing. We will see that there are also some compromises between these two kinds of slicing solutions.

### 2.1.1 Static slicing

The slice of a program is called a static slice when the algorithm performs without any information about the values of the input the program will receive. In this case the algorithm will need to preserve the interesting behavior of the original program for any possible input, thus producing a relatively big program slice.

### 2.1.2 Dynamic slicing

The slice is produced dynamically if the program slicing tool is aware the values of all the input variables of the program. This kind of slicing will be much more aggressive and produce much smaller, and thus much more readable slices. These slices are more or less a selection of the statements that were executed while running the program for the given input. The output program will not necessarily produce the same results for other input values. This kind of slicing is used mostly for debugging purposes.

### 2.1.3 Quasi static slicing

Dynamic slicing is often preferred because of its smaller slices, but in some cases it is not possible to know the value of all input variables. Some algorithms can still benefit from the information known for a few variables, these are called quasi static slicing algorithms. Quasi static slicing is a mix of static and dynamic slicing. When the values of some input variables is not provided, it will behave like a static slicing algorithm and produce a result that will preserve the properties of the original program for any possible value of the variables; when the input values are known, it will use them to eliminate some branches of the program and produce a smaller result. Quasi static slicing can be compared to partial evaluation in some functional languages.

### 2.1.4 Conditional slicing

Conditional slicing can be seen as a generalisation of the previous slicing algorithms. Instead of giving value for the input variables like for dynamic slicing, the developer will provide conditions on the values. With no condition at all on the values, this will fall back in a standard case of static slicing. With very strict conditions, like specific values for some or all input variables, this will fall into the dynamic or quasi static slicing cases explained above.

Conditional slicing will be of particular interest if the developer knows some specific information. For example, if the developer know that during the execution of a program an input variable $i$ will always be assigned a positive value, it is possible to give a condition like $i > 0$ in the slicing criterion. The resulting program will only handle positive values of $i$ and should be significantly smaller if some processing was done in case of a negative value of $i$.

## 2.2 Forward or backward slicing

A program slice can be computed either forward or backward with respect to the slicing criterion:

- A forward slice is a slice containing anything that depends on the code at the slicing criterion. A forward slice is ideal to have an overview of the potential impact of a modification at some point in the code.

- A backward slice is computed the other way around: such a slice will contain any code that affected the computation the values at the slicing criterion.

## 2.3 Scope of the slicing

All program slicing algorithms try to reduce the size of a program, but their effect is not on the same portions of the program. We will present a few areas that can benefit from program slicing.

### 2.3.1 Statement slicing

The first obvious candidate for slicing is the executable code. Most program slicing tools will try to reduce the number of executable statements.

This kind of slicing can be both inter-procedural or intra-procedural. These slicing algorithms work with dependency graphs and apply graph reachability algorithms on them to determine which statements are needed and which statements can be deleted.

There has been a lot of research on this topic to improve the quality of algorithms and deal with specific details of various languages, like pointers for C code (Livadas and Rosenstein, 1994), object oriented software (Chen and Wang, 1997; Larsen and Harrold, 1996), etc. . .

### 2.3.2 Declarations removal

While deleting executable statements seems at first sight the most attractive kind of slicing, it can also help developers to have tools that will reduce the number of declarative statements.

For example, if a developer has to look at a preprocessed file to understand a bug in the expansion of a macro, the file will be huge because there will be thousands of unused declarations coming from the included header files. Keeping the number of declarations to a minimum will dramatically reduce the size of the file that the developer will have to look at to debug the faulty macro.

### 2.3.3 Slicing class hierarchies

Some programing paradigms or some languages have specific slicing needs. For example, when dealing with C++ slicing, the ability to slice class hierarchies is welcome. Understanding class hierarchies can for example allow the slicing program to remove unneeded inheritances. Removing some unneeded inheritance dependencies will sometimes lead to remove whole classes from the program slice, and to produce a much smaller slice.

Slicing class hierarchies in C++ is a complicated problem: the slicing algorithm (Tip et al., 1996) needs to carefully follow the lookup system of C++, and take into account virtual and non virtual inheritance.

### 2.3.4 Unexplored areas

Some aspects of program slicing are still left mostly unexplored. Especially, current program slicing algorithms do not deal with preprocessed or templated source code, they all work on the preprocessed source code.

Being able to undo the action of the preprocessor after the execution of a program slicing algorithm would permit some interesting new applications:

For some applications, being able to go back to the original source file and add in it preprocessor directives to ignore the parts of the file that were not part of the slice would be very useful.

It would also be great if when undoing the preprocessing it was possible to notice that the content of a whole included file was completely useless, and to remove the corresponding `#include` preprocessor directive.

In chapter 4 we will focus only on static slicing because it is the kind of slicing we want to achieve with Transformers.

# Chapter 3

# Applications of Program Slicing

In this chapter, we will present a few use cases which will lead a programmer to want program slicing tools.

The common applications are split in two sections, depending on whether the resulting code is intended to be read by a programmer or to be executed. And finally, we will present a few emerging applications of program slicing.

## 3.1 Code readability

People working on the code of large-scale software projects spend much of their time reading code written by other people. These people actually spend much more time reading code than writing new code.

Sometimes the code is easy to understand, either because it was well written and well documented, or because the programmer already spent some time working in this area of the project not so long ago.

Unfortunately, reading some pieces of code can sometimes turn into a frustrating time consuming hard-work for the developer. This happens when the developer is confronted to some unknown poorly written, under-documented or messy source code.

Human reading and understanding of source code is slow and developers often have to read a lot of code before finding what they were actually looking for. To improve their productivity, developers are interested by tools outlining the code of interest in the context of what they are currently doing; this will reduce the amount of code the developer will actually need to read.

### 3.1.1 Debugging

Debugging is a common case that will make reading a huge amount code a necessity. Typically, the developer will notice that at some point in the program, some variables have unexpected values. Figuring out how these values are computed may not be trivial, program slicing can help a lot with this.

Let us just define the point of the program were the problem was detected and the variables containing unexpected values as a slicing criterion and execute a program slicing tool. Hopefully, this will produce a much shorter program that the developer should be able to read quickly. Once the problem is spotted in the program slice, it is easier to fix the bug in source code of the original program.

Program slicing used as a debugging tool is even more powerful if the developer can provide a specific testcase that leads to the bug and use a dynamic slicing algorithm. This way, the programmer will not need to read all the code related to computing the value, but only the code that computes the specific bogus value and thus will save extra time.

### 3.1.2 Understanding an area of functionality

When a developer arrives on a project and needs to rework, improve or finish the code providing some functionality to the application, the first thing to do will be to learn where this code lives in the project. If this is not documented, the usual way to learn this is to go through some directories of the source code tree, open some random files, read quickly to have an idea of the purpose of the file, go somewhere else and try again until you the desired code is found. Of course, in some cases, common search tools like `grep` can speed up the process but this will work only if the project does not contain several functions or types with the same names in different files.

### 3.1.3 Quality audit of the code

A common task requiring humans to go manually through huge amounts of source code is when doing an audit to ensure that the code complies with some desired properties.

Let us take two examples and show how program slicing tools can help for these specific cases.

#### Estimation of modularity

Modularity is often wanted when developing software because it reduces maintenance costs and will improve reusability for future projects. Unfortunately, it is not easy to check that the modularity was actually respected in the program. This problem is particularly true when a company outsources the development of some software components and wants to check the quality of the code that was produced before paying for the development.

Auditing the code manually is expensive and error prone. Slicing tools can help here either by simplifying the work of the person doing the audit, or even by automatically producing reports.

It is indeed possible to take more or less random slicing criterions in the source code and to compare the size of the resulting slice with the size of the full program. When applying this process, a code with a good modularity is likely to often produce small slices. A messy code breaking modularity principles however will very often produce slices that are almost as big as the full original program. Following this process several times and taking into account the average size of the obtained slices is an easy way (in the sense that it can be fully automated) to roughly estimate the modularity of a program.

#### Ensure security of sensitive data

When a program deals with sensitive data, an audit of how the data is processed may be required. This audit requirement means that someone will have to read and understand the code to certify that the code will not leak any sensitive data. This task is tedious, and the longer a program is, the more likely it is that an error will go unnoticed.

Program slicing can help a lot with this: setting the sensitive values and the point where they are set as the slicing criterion and computing a forward slice will outline the code that accesses directly or indirectly the sensitive data. By reducing the size of the program to analyse, it will simplify the process and thus make the results of the audit more reliable and cut costs at the same time.

Checking the modularity (as described previously) and that computations based on the sensitive data are located in a small area of the program are also safe things to do.

## 3.2 Downsizing

Developers often use libraries in the the process of creating software. Libraries are very attractive because they allow developers to reuse easily some code that has been written for another software project. Reusing code instead of rewriting it cuts development times and costs. Also, using stable, well tested and documented libraries rather than reinventing the wheel each time it is needed prevents bugs from being inserted by a distracted developer getting bored by the trivial code he has to write again and again.

A drawback of libraries is that the more featureful they are, the heavier they get. Usually this does not really matter because the libraries are installed as part of the operating system, they are shared in memory between several applications, and the cost of depending on a library providing unused fonctionalities is pretty small.

Nowadays disk and memory space are becoming cheaper than developer time so people tend to ignore the space wasted by some libraries. However, there are some people who care about the binary size of their application.

We can take people who distribute their applications on the Internet and who have to pay for the bandwidth as a typical example: they care about the size of the installation package users will download and they want to ship as little dependencies as possible.

Developers of applications intended to work on small devices like cell phones or robots care even more about the binary size of their applications. When embedding an application in these kinds of devices, storage space is expensive, so keeping the library sizes to a minimum is a requirement.

Another case where program slicing is desired is for web applications. More and more web applications use big AJAX framework. Using these frameworks can significantly reduce the development time of complex web applications, but they are also very slow to load, especially if the visitor has a poor Internet connection. Lots of companies currently use some kinds of JavaScript Compressors to downsize the JavaScript library sizes, but removing completely the unused portions with a program slicing tool is even better.

## 3.3 Emerging applications

We can anticipate that some applications of program slicing will become increasingly useful in the future. We will give two examples.

### 3.3.1 Test coverage

An increasing number of software development teams switch to a test-driven development model: each time they change something in the code of the application they develop, they add some unit tests to the test suite to make sure the feature they added will not get broken or the bug they fixed will not regress. Each time someone does a check in, the test suite is automatically executed to ensure that everything still works as expected and this validate the changes.

This test-driven development model increases the confidence that developers and testers have in the unstable builds of the applications. This way, it is possible to know faster when something is wrong and to only give usable builds to testers. As the unstable builds gets increasingly more stable, more testers try them, and regressions that could not be spotted by the test suite are more likely to be reported quickly.

To make this approach work, most of the source code should be checked by the test suite. Ideally, each line of code should be executed at least once when running the test suite. Even if executing each line of source code while running the test suite will not prevent bugs to appear when there are special cases that the developers did not expect, it will at least ensure that no big piece of code will get badly broken without anybody noticing it.

Program slicing tools can help to analyse the test coverage of the test suite (Binkley, 1998; Chen et al., 2003). To get an idea of how well the code base is covered by the test suite, it is possible to create a dynamic slice for each testcase of the test suite, and then to compute the union of these slices. Finally, comparing the length of the produced program slice with the length of the full program gives an idea of how mature the test suite is. The differences between the computed slice and the full program indicates which areas of the code developers willing to improve the test suite should focus on.

### 3.3.2 Parallelization

With the emergence of new multi-core CPUs, many legacy applications will need to be rewritten to use threads and thus work on multiple cores at the same time.

Program slicing technologies can help here, by identifying pieces of code that have little or no dependencies with each other, and separating them in different slices. With little adaptation work, it should then be possible make these slices run simultaneously on separate threads.

This technology is not mature yet, but we can believe that with the increased pressure to make applications work on multiple threads, more researcher will focus on this area and will eventually manage to make it usable.

Chapter 4

# Implementation with Transformers

## 4.1 Context

### 4.1.1 Transformers

Transformers is a C++ transformation framework based on the Stratego/XT transformation toolkit (Bravenboer et al., 2006).

Transformers aims at providing all the tools needed to do source to source transformations on C++ code. This means that Transformers' tools take C++ code as input and will output the same source code with some transformations applied.

The C++ transformation pipeline is roughly in three big steps:

**Parsing** First we need to parse the input source code. In this process Transformers will first make use of a preprocessor. We have developed our own preprocessor, revcpp (Sigoure and Hocquet, 2008), to ensure it is possible to undo these preprocessing later and get back to the original source code.

Once the code is free of any preprocessing directive, we call SGLR (Visser, 1997) which produces an ambiguous output (a parse forest instead of a parse tree) because the C++ grammar is ambiguous.

Then, we need to apply a disambiguation process to our parse forest, so that we keep only a parse tree. This disambiguation is done using an attribute grammar system (David, 2004).

The parse tree is now ready for the next step: transformations.

**Transformations** Transformation tools should be written in Stratego/XT. They take a parse tree in the asfix (van den Brand and Klint, 2007) format as input, apply transformations rules on it, and then output the tree in the same asfix format.

**Pretty printing** Once the program represented as a parse tree has been transformed, we need to call a pretty printer to get back to a text representation of the program.

We can if needed undo the preprocessings that were made at the first step with revcpp.

### 4.1.2 Motivation

The development of the tools that are part of Transformers has been going on for a long time. While the disambiguation is not completely finished for some specific cases, we believe it is already usable.

To expose the power of Transformers, we need to have some sort of 'killer applications' that we can show to people. These applications are more or less demonstrations of what we can expect to do with transformers.

The requirements for such an application are as follow:

- They should be relatively quick to implement.
- Their usefulness should be obvious.
- They should be simple to use.
- They must be difficult to implement without Transformers.

We believe that creating a C++ program slicing tool with Transformers meets all these requirements.

Out goal here is to create some easy to use software to extract a region of interest from a large program. We will present in the next section how we can implement that with our tools.

## 4.2 Proposed implementation

### 4.2.1 Usage

Setting up a working Stratego/XT environment and installing Transformers tools is not really trivial. While it is worthwhile for someone who plans to use these tools a lot and develop his own transformation tools based on them, to make a short demonstration of the power of Transformers, it is nearly a show stopper.

We decided that to make this program slicing demo, the user interface should be in the form of a web service. Some previous work has already been done on creating web services with Stratego/XT (Bravenboer, 2003). Providing a web interface should make this demonstration readily usable by random users looking around on the transformers web site, and should attract them to read more about Transformers.

To use the tool, the user will go to a web page with a form, upload a C++ source file, select the region that is interesting and validate to start the processing. Our server should return the shortest valid program still containing the region of interest selected by the user.

### 4.2.2 Limitations

The first versions of our tool, even if already useful, will not compete with the State of the Art C++ program slicing algorithms.

There are some limitations in the implementation we will provide:

- We will not handle the preprocessing. The user will have to do it before sending the file to our system. It should be possible to use revcpp to preprocess the macros, but as we will have a single input file, it would not be possible to any thing for `#include` preprocessing directives.

- We will not do intra-procedural slicing.
- We will not handle templates (at least at the beginning)
- As Transformers does not compute types yet, our slices will tend to be larger than they could be: we will not be able to eliminate any overloaded method if at least one of them is used.
- The first implementation will not handle namespaces, handling this later should be pretty straight-forward though.

### 4.2.3 Algorithm

The algorithm that we will use is quite simple:

First we will compute a list of code blocks. A block will be either a declaration or a function/method.

For each block, we will keep some information, like the name and the basic type (if it is a type definition, a function declaration, a function definition, … later we will keep the full type).

We will also compute a list of backward dependencies: in case we need to include this code block in the final slice, we need to know which are the other code blocks this code block depends on.

The list of code blocks and there dependencies builds a dependency graph. We will then apply a graph reachability algorithm: we will mark all the blocks that are reachable from the blocks that are located in the region of interest.

Finally, we will pretty print only the blocks that are marked, and get rid of everything else.

This algorithm can be seen as a mark & sweep algorithm like those that are used for memory garbage collection.

### 4.2.4 Implementation details

#### Computation of dependencies

To compute the dependencies, we need to have a parse tree without ambiguities, so the first thing to do is to apply Transformers' standard disambiguation system using attribute grammars.

To detect dependencies, we do a tree traversal on the non ambiguous parse tree. Each time we visit a node that starts a new code block, we will assign a new unique identifier for it. Then, while visiting sub nodes, each time a lookup in the symbol table was needed while parsing and disambiguating the tree, we have a dependency. We can store the list of dependencies alongside the unique identifier of the code block.

#### Client or server side processing

We are not sure yet of how much processing will be done on the server side and the client side.

One solution is to do the whole processing on the server side and return a pretty printed program.

In this case, once we have the dependency table, we have to execute a graph reachability algorithm inside our parse tree, we can mark the useful node and ignore branches that are not marked while pretty printing.

Another solution would be to only parse and compute dependencies on the server, and then extract the exact slice on the client side, so that changing the region of interest does not require to re-parse. This later solution would allow the user to dynamically change the region of interest and see the result very quickly without waiting for a server interaction.

In this case, the server needs to send to the client the location of the various code blocks and the list of dependencies for each of them.

# Chapter 5

# Conclusion

We have shown that Program Slicing techniques have several useful applications. The two usual use cases are to help a developer to understand the source code of a program, or to reduce the size of a program. Depending on the needs, various slicing algorithms exist.

We explained how we will build an easy to use online C++ slicing tool based on Transformers. Even without being a full featured slicing tool, this will be a good demonstration of the power of the Transformers framework.

# Chapter 6

# Bibliography

Binkley, D. (1998). The application of program slicing to regression testing. *Information and Software Technology*, 40(11-12):583–594.

Bravenboer, M. (2003). Connecting XML processing and term rewriting with tree grammars. Master's thesis, Utrecht University, Utrecht, The Netherlands.

Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2006). Stratego/XT 0.16. Components for transformation systems. In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*, Charleston, South Carolina. ACM SIGPLAN.

Chen, J.-L. and Wang, F.-J. (1997). Slicing object-oriented programs. In *Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC'97/ICSC'97)*.

Chen, Z., Xu, B., and Yang, H. (2003). Test coverage analysis based on program slicing. In *IRI*, pages 559–565.

David, V. (2004). Attribute grammars for C++ disambiguation. Technical report, LRDE.

Larsen, L. and Harrold, M. J. (1996). Slicing object-oriented software. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 495–505, Washington, DC, USA. IEEE Computer Society.

Livadas, P. and Rosenstein, A. (1994). Slicing in the presence of pointer variables.

Lucia, A. D. (2001). Program slicing: Methods and applications. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE Computer Society Press, Los Alamitos, California, USA.

Sigoure, B. and Hocquet, Q. (2008). revCPP a reversible C++ preprocessor. Technical report, EPITA Research and Development Laboratory (LRDE).

Tip, F., Choi, J.-D., Field, J., and Ramalingam, G. (1996). Slicing class hierarchies in c++. *SIGPLAN Not.*, 31(10):179–197.

van den Brand, M. and Klint, P. (2007). Aterms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology*, 49(1):55–64.

Visser, E. (1997). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam.

Weiser, M. (1984). Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357.

Weiser, M. D. (1979). *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, USA.