



Cost Efficient Dependable Electronic Systems

Set and Get in AspectC++

Author	Johan Magnusson
Document Id	013
Date	1 February 2006
Availability Status	Public Final

CHALMERS



Set and Get in AspectC++

JOHAN MAGNUSSON

Master's Thesis

Computer Science and Engineering Program

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Division of Computer Engineering
Göteborg 2006

All rights reserved. This publication is protected by law in accordance with "Lagen om Upphovsrätt, 1960:729". No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of the authors.

© Johan Magnusson, Göteborg 2006.

Abstract

The purpose of this thesis is to examine and implement set and get joinpoints for aspect-oriented programming in the C++ language. These types of joinpoints are needed to be able to implement certain techniques, for example different fault tolerance mechanisms.

A basic introduction to what aspect-oriented programming is and how it can be used is given. The reasons behind why the set and get joinpoints are important is also discussed.

A few problems that restrain the implementation of set and get in C++ is highlighted, the main of which is that C++ uses pointers which makes it harder to detect when some data is accessed.

An implementation that does not handle pointers is presented. This implementation has been performed by extending the open source AspectC++ compiler.

Finally a possible solution to the problem introduced by pointers is presented and discussed. The solution consists of an adaption of a points-to algorithm to the problem. Alternative solutions are also briefly discussed.

Sammanfattning

Målet med detta examensarbete är att undersöka och implementera set och get joinpoints för aspect-orienterad programmering till programmeringsspraket C++. Dessa typer av joinpoints behövs för att implementera vissa tekniker, till exempel olika typer av mekanismer för feltolerans.

En grundläggande introduktion till vad aspekt-orientering är och hur det används ges. Varför set och get joinpoints är viktiga diskuteras också.

Några problem som hindrar implementeringen av set och get i C++ belyses. Det huvudsakliga problemet är att C++ innehåller pekare vilket gör det svårare att avgöra när åtkomst av en viss data sker.

En implementering som inte hanterar pekare presenteras. Denna implementering är gjord genom att utöka open source kompilatorn AspectC++.

Slutligen introduceras och diskuteras en möjlig lösning på pekarproblemet. Denna lösning består i en anpassning av en points-to algoritm. Alternativa lösningar diskuteras också kortfattat.

Acknowledgements

I would like to thank Ruben Alexandersson who initiated and supervised my thesis.

My sister Stina Magnusson was a great help in correcting my english, which I thank her for.

Thanks also go out to Erik Bengtsson who provided good constructive criticism on the language and contents of this report.

Contents

1	Introduction	4
1.1	Purpose	5
1.2	Limitations	5
2	Aspect-Oriented Programming	6
2.1	Background and Motivation	6
2.2	Terminology	8
2.3	Aspect Oriented Fault-Tolerance	9
3	Problem Analysis	11
3.1	Set and Get Occurrences	11
3.1.1	Values	11
3.1.2	Pointers and References	11
3.1.3	Operator-overloading	13
3.2	Comparison with Java	14
4	Implementation	17
4.1	The AspectC++ Compiler	17
4.2	Specification	18
4.2.1	Pointcut Syntax	18
4.2.2	Joinpoint Creation	19
4.2.3	Limitations	19
4.2.4	Joinpoint Information	20
4.3	Modifications and Implementation Details	20
5	Points-to algorithm	23
5.1	Study of Points-to Algorithms	25
5.2	Goal	26
5.3	Limitations	27
5.4	Algorithm	27
5.5	Invisible Variables	32
5.6	Effectivity	36

6	Conclusions	37
6.1	Extensions and Future Work	37
A	AspectC++ Source Code Modifications	40

Chapter 1

Introduction

Aspect-oriented programming (AOP) can not be considered a new technique anymore with 10 years behind it, and as always the ideas dates back further. With competent tools available it is starting to heat up as a practical solution to many common cross-cutting concerns which crops up in larger software systems.

Fault-tolerance is one among these cross-cutting concerns which has been considered and implemented. One significant advantage of implementing fault-tolerance mechanisms in AOP is that it becomes feasible to replace hardware based fault-tolerance with software that reduces the cost of hardware as well as the physical weight of the system. With previous software techniques the implementation of fault-tolerance results in complex systems with fault-tolerance code scattered throughout the system making it expensive to both construct and maintain. Some technical obstacles yet remain though, as the most mature tools for AOP is available for the Java programming environment only; an environment which is not well suited for embedded systems with low processing power. A better candidate for these situations would be AOP for C++.

Many fault-tolerance mechanisms is centered around monitoring data, for example verifying computational results or making sure a value keeps within a certain range. One such monitoring mechanism is the recovery cache, which tracks changes to data and allows for rollback to a previous program state. The recovery cache has been implemented successfully using AOP in Java giving a versatile and pluggable solution [11]. These mechanisms can be implemented in the mature Java environment since it has support for detecting reading and writing of variables, something that the tools for C++ lack.

The reason that there is no support for detecting reading and writing of variables in C++ is because it is not always decideable when the reading and writing of a certain variable occur. This leaves the tools either without support for the feature or with an incomplete support which works in some

cases but not in the general case. Currently the developers of these tools have other priorities. The problems with deciding when a variable is accessed arise because of the rich support for pointers and references which is available in C++, the tradeoff from this power is a more complex and difficult language to analyze.

Deciding where in a program a certain value is accessed with a pointer can be done by performing a so called points-to analysis. The traditional application for these algorithms is optimization of a program at compile-time, but the result can be used in many other areas. Many such points-to algorithms exist, but still the most common compilers use simpler versions to perform their optimizations. This hints at the complexity of the task. The problem with the algorithms is usually that they are either very inefficient or imprecise.

1.1 Purpose

The purpose of this thesis is to examine the problems surrounding the creation of set and get joinpoints when implementing AOP for the C++ programming language. The possibility to implement these joinpoints in a compiler should also be considered.

The main goal of the implementation part of the thesis is to implement support for set and get in the AOP compiler AspectC++. The resulting compiler is to be used as an experimental environment to evaluate different mechanisms that require the set and get constructs.

1.2 Limitations

As the thesis has a limited time frame some limitations of the work scope is needed. These limits were not clear from the start but as the work progressed a set of tasks were decided as a reasonable result.

The implementation of set and get in the AspectC++ compiler is limited to work with named access of variables, disregarding pointers. Such a simplified solution is suitable for experimental use.

Efforts on a solution to the problems that occur when pointers are considered are limited to reasoning, no actual implementation is performed. These efforts also focus on the possibility of using a points-to algorithm to improve the brute-force solution to the pointer problem.

Chapter 2

Aspect-Oriented Programming

2.1 Background and Motivation

Figure 2.1 gives an indication of the popularity of the aspect-oriented programming research field, this popularity has a potential to keep growing as the usefulness of AOP can be examined in context of most fields in computer science. AOP can also be implemented in most programming languages where every language presents its own opportunities as well as problems.

The term aspect-oriented programming was introduced in 1996 by researchers at Xerox. The technique was created to ease an unconventional type of modularization, namely to separate cross-cutting concerns from the primary concerns of a system. [19]

Cross-cutting concerns means functionality that affects large parts of the program, without really being the main task of the code. An example is logging bank transactions, where the task is to perform the transaction but a side task is to log when it happens. By separating the implementation code which solves the cross-cutting concerns and the code which addresses the main functionality the system becomes better structured where every part of the code can focus on its real task. The following sentence summarizes this in an effective way:

Instead of spreading the code related to a design decision throughout a program's source, a developer is able to express the decision within a separate, coherent piece of code. [25]

The idea with aspects is that the code affected by the aspects are oblivious of the influence [13]. Experiments show that it is not always easy to design an aspect-friendly system from the start, instead changes may need to be performed to allow effective AOP [21]. Making changes to accommodate

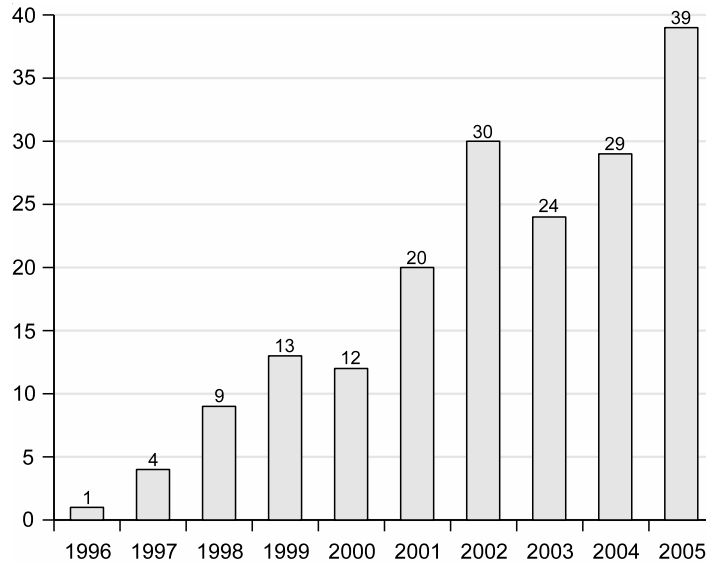
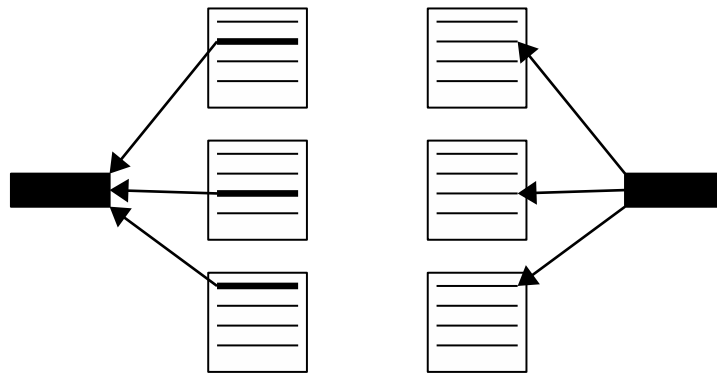


Figure 2.1: The number of publications found in the ACM Digital Library per year with “aspect-oriented programming” either in the title or as a keyword. Source: ACM Digital Library [1]



(a) The subroutine is explicitly called from several places in the program modules resulting in scattered code.

(b) The aspect implicitly invoke the cross-cutting code at the appropriate places in the program.

Figure 2.2: Implementation of a cross-cutting concern with a subroutine (a) and an aspect (b).

aspects breaks the idea of obliviousness in some sense, but this is not a big issue since the concerns are still separated.

AOP is not a replacement for old types of modularization, like object-oriented programming (OOP) and sub-routines. Current AOP systems use these program structures to introduce the aspect-code in a controlled manner [13]. This means that the major parts of the program will most likely be written in a traditional style with a few aspects to help with the most typical cross-cutting concerns, in the future it could be that programs are constructed without any OOP or similar but by combining a number of aspects to form the whole of the program.

An important difference, which has already been touched upon, between using sub-routines and aspects, is that sub-routines have to be called explicitly while aspects are called implicitly (see Figure 2.2), this decoupling is at the heart of AOP. As is often the case; a strong point of a technique can also be a weak point depending on how it is viewed. On the one hand the implicit invocations allows a specialized programmer to write the code which handles the main functionality and another programmer to do the cross-cutting functionality [13]. On the other hand the programflow is harder to follow with implicit invocations, which can cause problems in debugging and program understanding [20].

The most mature implementation of AOP to date is the AspectJ [10] compiler for Java. Since this compiler implements joinpoints for set and get, a few comparisons will be performed to find out why it is harder to implement these joinpoints in C++ as compared to Java.

2.2 Terminology

AOP brings along a number of important key words that are essential to know and understand to be able to comprehend texts about AOP. This section serves as a glossary for the most important of these key words.

Advice is the code which implements the functionality to handle a cross-cutting concern.

Aspect is a collection of pointcuts and advice which the programmer wants to group, usually to handle one cross-cutting concern.

Cross-cutting concern is a concern which affects many parts of the program, it cuts across the system. A few examples are logging, authentication, verification, security and fault tolerance.

Introduction is used to modify not only the behaviour of the program but also the structure of it. For example, to add a member function or field to a class is an introduction.

Joinpoint is a point in the target program exposed to aspects. This is where the programmer can choose to insert additional functionality.

Dynamic joinpoint is a joinpoint which can happen during execution, like the execution of a function.

Static joinpoint is a joinpoint which belongs to the program structure, where introductions can be made. A class in C++ is an example of a static joinpoint.

Pointcut is a selection of joinpoints. This selection can be connected to an advice to insert functionality at all the joinpoints this pointcut selects. An example of a pointcut expressed in natural language is “All functions with three parameters”.

Weaving is the process of combining aspects with the main code of the program, or other aspects, to form the final system.

Proceed is the action of continuing the execution of a program after an around advice has been executed.

A typical usage of AOP is to specify a *pointcut*, that captures the interesting *joinpoints* in the program and one or more *advices* or *introductions*, that solve the *cross-cutting concerns* at hand, in an *aspect* and finally *weave* it with the target program.

2.3 Aspect Oriented Fault-Tolerance

The largest gain from using AOP is received when solving wide concerns that span as much of the system as possible. The narrower the aspect the more specific pointcuts have to be constructed, resulting in more work for less functionality. Narrow aspects will also require more knowledge about the underlying system putting more strain on programmers and restricting reusability. Aspects solving wider concerns, on the other hand, will allow for the highest degree of decoupling from the base system. These wide concerns was the reason AOP was conceived in the first place.

One such wide concern is fault-tolerance. Fault tolerance literally means that a system tolerates faults. If an error occur the system should if possible handle the fault without any consequence, or otherwise gracefully degrade the system making the most of the situation. [2]

AOP would be ideal to use when implementing fault-tolerance since the fault-tolerance is no end in itself; it is a typical cross-cutting concern adding functionality on the side of the main task of the system. Implementations of different types of fault-tolerance mechanisms (FTM) exist, for example for Java. To enable these FTMs the monitoring of variable access is essential since they build upon the idea of inspecting the system state when it

changes. Since FTMs would be beneficial to implement in C++, because of its usage in embedded systems where fault-tolerance often is needed, the AspectC++ should implement constructs to detect these system state changes. This is the reason for trying to implement set and get in C++ despite the problematics involved. [11]

Chapter 3

Problem Analysis

3.1 Set and Get Occurrences

3.1.1 Values

The operations `set` and `get` correspond to the reading of a value and the writing of a value respectively. In Figure 3.1 a simple example of set and get with values is shown. Support for this kind of set and get is possible to implement without any problems, one implementation is presented in Chapter 4 with the limit that it works only on member fields.

```
int x, y;  
x = 0; // This line contains a set joinpoint  
y = x; // Both a set (y) and a get (x)
```

Figure 3.1: Simple form of set and get.

3.1.2 Pointers and References

In C++ it is possible to use references to variables. This way a programmer can modify the same value in memory by using different names. Consider the program in Figure 3.2; after the first two lines both `n` and `alias` will be backed by the same memory location. What if there is a pointcut defined that will trigger on the setting of the variable `n` - then it is not clear if the `alias++;` will trigger that pointcut. The problem of not knowing what locations a name is connected to is called the alias problem.

A similar problem arises with pointers, as is shown in Figure 3.3. There is a small difference in that `alias++;`, in Figure 3.2, could trigger a pointcut defined for the variable `alias` while there is no such joinpoint in Figure 3.3. The statement `(*pointer)++;` is simply a modification of the memory that pointer is pointing to, the pointer itself is not set.

```

int n = 0;
int &alias = n;
alias++; // This is a set (and get) on alias,
        // but the value of n also changes

```

Figure 3.2: The alias problem.

```

int n = 0;
int* pointer = &n;
(*pointer)++; // This will update n, but how can we tell?

```

Figure 3.3: The alias problem with a pointer.

In both examples from Figure 3.2 and 3.3 it is possible to decide at compile time that `alias` and `*pointer` are aliases for `n`. But this is not always the case as is shown in the example in Figure 3.4. At the end of this program the `pointer` can point either to `one` or `two`; which one it points to will depend on the random value returned from `rand()`.

```

int one = 10;
int two = 20;
int* pointer;

// Generate a number between 0 and RAND_MAX
srand((unsigned)time(0));
int random = rand();

// Make a 50/50 on which variable to point to
if (random < RAND_MAX/2) {
    pointer = &one;
}
else {
    pointer = &two;
}

```

Figure 3.4: Randomized assignment of pointer.

As seen in Figure 3.4 it is not always possible to compile time to determine which variable a pointer will point to when the program executes. This forces us to consider an approach where memory locations are tracked and compared at runtime to determine when a variable is read. The question is what kind of performance hit this solution would bring, and how much the performance can be improved by analysis [3].

Program	Lines with address taken of a field
Php 5.0.5	0.5%
AspectC++ 0.9.3	0.007%
Gtk+ 2.8.6	0.2%
Qt 4.0.1	0.2%

Table 3.1: A measure of the usage of taking the address of a field.

In an object-oriented language such as C++ or Java the program state is mostly held in the member fields of objects, which makes it most useful to enable set and get joinpoints on the access of member fields. This is how the implementation in AspectJ works as well as the one described in Chapter 4. In light of this the alias problem has a diminished role, since it is not good practice to access member fields by a pointer. An example of this is if a member function returns a pointer to a private field; the caller can then access the field outside of the class breaking the encapsulation. It is also important to note the difference between having a pointer *to* a member field and having a pointer *as* a member field. The alias problem is only present when having pointers to member fields. A pointer member field is just a value that works as any other. The difference between these two cases is illustrated in Figure 3.7 and 3.6. As an illustration of the small usage of accessing member fields by pointer Table 3.1 shows the number of times the address of a member field is taken in a few open source projects.

There are situations where set and get would be useful for local as well as global variables and member fields. For most cases member fields are enough; global variables is a discouraged technique and creating pointcuts on local variables would require very detailed knowledge of the target program. One problem that could be solved if local variables were tracked is memory management. For example a reference counting garbage collector that is transparent to the target program could be very useful.

In Chapter 5 a solution to the alias problem is discussed.

3.1.3 Operator-overloading

The default behaviour when assigning one variable to another in C++ is a copy operation. The natural thing to do would hence be to trigger any matching set pointcuts whenever an assignment is made.

The choice of what to do when there is an overload of the assignment operator present is not as natural since the overloading function can take any action, not at all limited to assignment. An operator overloading is actually a function in the class with a special name, and it is already possible to make pointcuts on these functions in AspectC++ [24].

One way to resolve the overloading problem is to trigger the set advice

only when there is no overload function present. Any class with an overloaded assignment operator would then be excluded from any set pointcuts. This way pointcuts for the `operator=` function can be added by the programmer when they actually perform assignments. This solution has the drawback that the program behaviour can change in an unexpected way if the programmer decides to implement an assignment overloading when there is a set pointcut in place.

The other option, to match any set pointcuts even if the class has overloaded the assignment operator, works fine as long as the assignment operator is not used in a non-standard way. This approach is more consistent since the behaviour does not change, but as the cost of the consistency is less power and flexibility since the programmer has fewer choices. Another problem with this solution is the double matching that can occur; if pointcuts exist on both set and the assignment operator, both of these pointcuts will match an assignment. The double matching gets worse when composite pointcuts, where the set is only monitored in a certain context for example, are used since the triggering of the advice code will be harder to predict.

3.2 Comparison with Java

The Java memory model consists of two kinds of data: primitive types and objects. The primitives are the basic integers, floating point numbers and single characters, while the objects are created from classes that the programmer can specify. Values of the primitive type is always copied when assigned to something, which implies that there can never be more than one name which refers to the same piece of primitive data. Objects on the other hand are always accessed by a reference; the data itself is stored in a memory space called the heap. So when assigning one object to another it is in fact only a reference which is copied, creating an alias. An example of this can be seen in Figure 3.5.

C++ has a more direct memory model with less abstractions. In C++ almost everything is a value: primitive types, objects and pointers are all values. This means that when assigning one variable to another in C++ the assigned value gets overwritten with a copy of the value to the right of the assignment operator. When creating a pointer the address of a variable has to be obtained, which can be done with the `&` or `new` operators. These operators return the physical memory address of the variable that can be assigned to a variable of pointer type. To access the value that the pointer points to is called to *dereference* the pointer, which is done with the dereference operator (`*`).

It is interesting to note that the alias problem described in Section 3.1.2 also exists in Java. As opposed to C++ aliasing does not affect when to trigger set and get. The difference between C++ and Java is that you can

dereference C++ pointers but not Java references. This difference means that the only way of accessing a field of a class in Java is by using its name. Figure 3.5 illustrates how a reference to a member field is created in Java and that this reference is then totally separated from the field. The joinpoints exist only where the field itself is accessed, not the actual data that the field references [4]. In contrast figure 3.6 shows how a pointer in C++ can modify the internal variables of an object via a pointer.

In C++ functionality similar to that in Java would be useful, but since C++ has the alias problem some variable access will not be detected by the aspect compiler giving a solution that is not optimal. In many situations it would be preferable to have a way of triggering on the access of a field from both named use and aliases. An example of a situation where this kind of mechanism would be useful is in runtime assertions. Such an assertion can be used to restrict the values allowed to be written to a variable. If the value of the variable is set using a named access or by following a pointer should not matter as it is still the same memory location that is set.

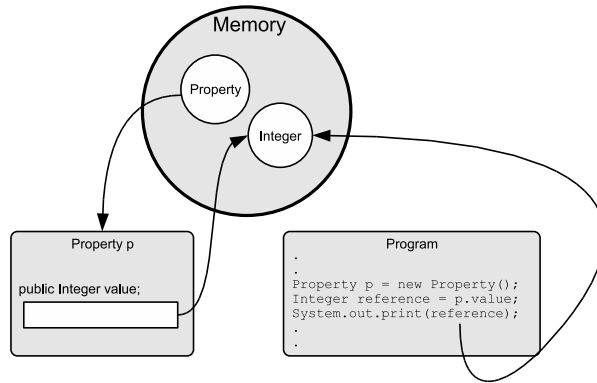


Figure 3.5: Referenced access in Java

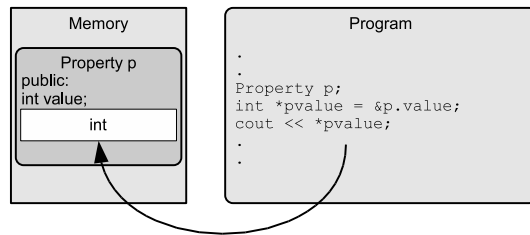


Figure 3.6: Pointer access to a field in C++, this scenario can not occur in Java.

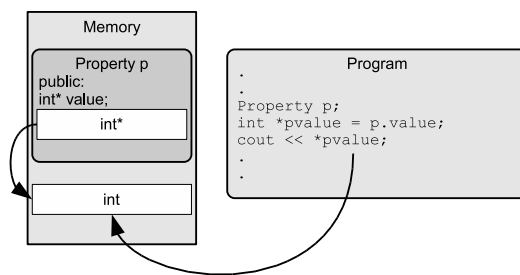


Figure 3.7: Copying of pointer in C++, this is the same situation as in Figure 3.5.

Chapter 4

Implementation

As seen in Chapter 3 there are a number of problems connected with the use of set and get in AspectC++. These problems do not mean that a simple implementation is impossible, just that it is non-optimal. In some cases a less powerful version that does not consider the alias problem would be useful, at least for experimental purposes. For real world applications a more robust implementation would be needed, whereas when experimenting more restrictions can be made on the target-programs that are tested. When creating test-programs the restrictions that should be applied can be found by looking in Section 4.2.3 on limitations.

This chapter describes an implementation of the set and get functionality without addressing aliasing.

4.1 The AspectC++ Compiler

The AspectC++ project [5] has implemented a compiler that is released as open source (GPL[6]). Version 0.9.3 of this compiler has been modified to allow pointcuts to be set on set and get joinpoints. This section describes the general structure of the compiler.

The compiler works by taking the program source code, that can contain aspects, and transform it by weaving in any advice declared in the source. The output from the compiler is standard C++ code that can be compiled with a regular C++ compiler. This kind of compiler is called a “source to source compiler”, it can also be referred to as a “preprocessor”. [15]

When making a compiler such as the one described here, it is not purposeful to create a new parsing system for the target language. This kind of system has most likely been created many times before. Instead, the AspectC++ project uses an open source system called PUMA[22], which provides lexing, parsing, semantical analysis and code manipulation for the C++ language. PUMA has been extended by the AspectC++ project to be able to also parse aspect specific code. [15]

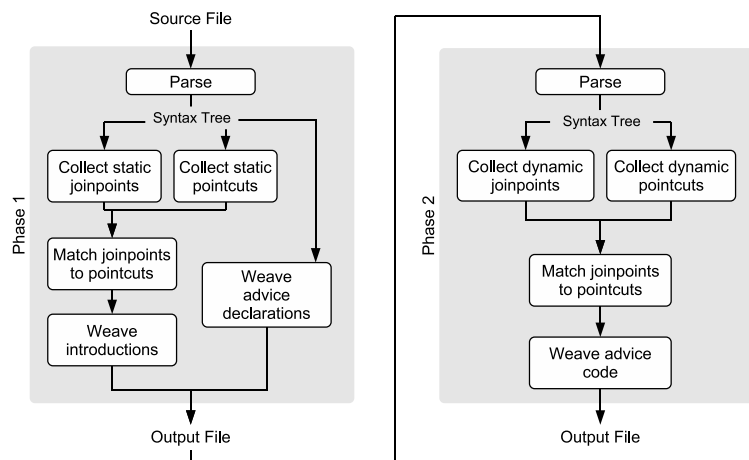


Figure 4.1: Basic flow of operations in the AspectC++ compiler

Figure 4.1 outlines the flow of operations in the compilation process, which basically breaks down into two phases.

The first phase takes care of all static joinpoints, so that all introductions are handled in this phase. All aspect declarations are also converted into regular C++, which is done simply by making every aspect a class and every advice a function.

During the second phase all dynamic joinpoints in the code are collected and matched against all pointcut declarations. For every matched pointcut any associated advice code is weaved before, around or after the corresponding joinpoint location.

4.2 Specification

This section specifies the additional functionality which has been added to the AspectC++ compiler.

4.2.1 Pointcut Syntax

The syntax for variable access is defined as

```
"type scope::name [const]"
```

```
"int aClass::%", any member field of type int in aClass.
```

```
"% ...::aField", the member field aField of any type in any scope.
```

```
"char ...::% const", any variable of type char that is declared const
```

4.2.2 Joinpoint Creation

How and when the joinpoint locations are created is covered in Section 4.3. The following criteria is used to determine where to create set joinpoints:

1. The statement is an assignment.
2. The left operand is
 - a simple name (`field`) that is a member field of a class
 - or a member reference (`object.field`)
 - or a member access by pointer (`objectpointer->field`).

A new get joinpoint is created when the following pattern is found:

1. The expression is
 - a simple name (`field`) which is a member field of a class
 - or a member reference (`object.field`)
 - or a member access by pointer (`objectpointer->field`).
2. The expression is not the left hand operand of an assignment.
3. The expression is not the operand of the address operation (`&`).

4.2.3 Limitations

The following has been considered but not implemented.

1. Other assignment operators than `=`, like `+=`, `-=`, `++` and `--` has not been implemented, they would work the same way as the regular assignment operator.
2. Set joinpoints are not created when variables are initialized.
3. Set and get joinpoints can only be created on member fields, not on local or global variables.
4. Setting and getting of a member field will only be detected if it is done by name; if a pointer is used it will go by unnoticed.

Why access by pointer is not implemented is discussed in Section 3.1.2. The situations where problems occur are when passing member values to a pass-by-reference function or when accessing a member field with a pointer, something that should not occur if proper encapsulation is maintained.

There is a fine line between initialization and assignment, this implementation considers them to be separate. It could be argued that a variable is written when it is initialized. It could also be argued that the variable is created, not written, when initialized.

4.2.4 Joinpoint Information

In advice code, information about a particular joinpoint location can be accessed using the `tjp`¹ data structure. The information contained in this structure is the same as for any other joinpoint. In addition a pointer to the variable being read/written is provided (`tjp->dest`). In case of a set joinpoint the value that is going to be written can be accessed with `tjp->source`.

It is not entirely straightforward what it means to do a proceed in an advice for a get joinpoint, or perhaps more puzzling what it means *not* to do a proceed. If no proceed is performed, the field should not be read, but if the field is not read, what value will be the result of the operation? The approach taken in this implementation is to say that an advice around a get joinpoint without a proceed has an undefined result, unless the result is set explicitly with for example `*tjp->result() = value;`. Another approach could be to raise an error when this situation arises. In AspectJ an advice can have a return type - this way around advice will require you to return a value. Maybe this approach would be suitable for AspectC++ as well, in which case it could be applied to around advice for other joinpoint types as well.

4.3 Modifications and Implementation Details

Since set and get joinpoints are dynamic - they occur during execution - the implementation of them was placed in the second phase of the compilation process.

To be able to collect joinpoint locations the AspectC++ compiler models a joinpoint location with a class. All different types of joinpoints inherit this class, which is called `JoinPointLocation`. Two sub-classes of `JoinPointLocation` has been created to model a set and get joinpoint location respectively: `JPL_Set` and `JPL_Get`. These classes contain information about the joinpoint. For example the syntax tree and the type of the variable being accessed.

The collection of joinpoints is performed by traversing the syntax tree recursively, creating instances of the right class when finding a tree which is a joinpoint location. For example, when finding a binary expression the operator is examined. If it is an assignment the left operand is examined. If the left operand is a field in a class then a new instance of the `JPL_Set` class is created. The syntax tree recursion is performed in a class called `TrackerDog` and the additions to create the set and get joinpoints has been added to this class.

All joinpoints are later matched against pointcuts, represented in the

¹tjp is short for 'this joinpoint'

compiler by subclasses of the `PointCutExpr` class. Two new classes are introduced to represent set and get: `PCE_Set` and `PCE_Get`. These classes can tell if a given joinpoint location matches that pointcut. The instances of these pointcut expression classes are created in a class called `PointCutEvaluator` and both inherit the `PointCutExpr` class. `PointCutExpr` has the same role as `JoinPointLocation` but for the pointcut expressions.

Code to take care of the actual weaving is added to the `CodeWeaver` class, which basically takes care of all weaving in the compiler. The code manipulation itself is performed by the PUMA system at token level, pasting strings before or after individual tokens.

When weaving advice code for a set or get joinpoint a so called wrap function is used. The code corresponding to the joinpoint is replaced by the invocation of the wrap function. The declaration of the wrap function is placed so that it will be accessible from the joinpoint and contain the calls to the before, around and/or after advice present. In case of a set joinpoint it will also contain the actual assignment.

Along with the wrap function the “this joinpoint” data structure is created. Set and get specific members of this structure are described in 4.2.4. The members of the data structure are created on demand, which means that if they are not used in the advice they are not created.

To be able to use the `tjp->proceed()` function in an around advice, yet another wrap function is introduced. This function contains the actual assignment in case of set or the assignment of the source variable to the result buffer in case of get. Just as with the members of the “this joinpoint” structure the proceed function is created only on demand.

The overall workflow is implemented in the `Transformer` class. This class has had two new loops introduced which iterates over the matched set and get joinpoints weaving them along the way.

To avoid nameclashes with the base program all weaved code has somewhat obfuscated names that are unlikely to be used by a human. To generate these names a few methods have been added to the `Naming` class. This class makes sure that the same object gets the same name every time it is obfuscated.

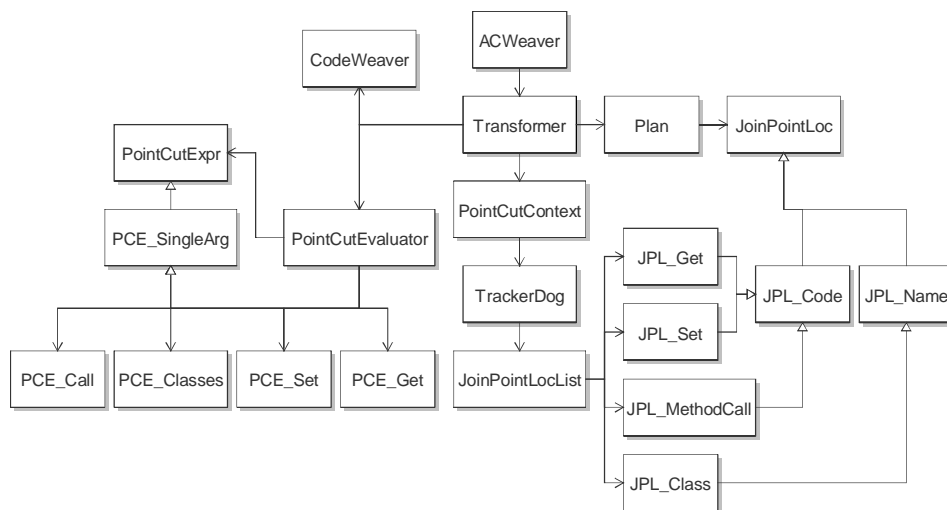


Figure 4.2: Much simplified UML Class Diagram of the AspectC++ compiler with a few set and get specific classes added.

Chapter 5

Points-to algorithm

As seen in Section 3.1.2 one big challenge to set and get in C++ is aliasing; a pointer can possibly point to different locations at different executions. It is also established that there is no way to statically determine which aliases will be present at run-time. What can be done is to run an algorithm that collects (or estimates) all possible locations where a pointer can point to when it is dereferenced. Using such an algorithm the number of runtime checks can be held to a minimum.

Not using a points-to analysis would make a very inefficient, but still functional, program. Without the analysis an approach would be to gather all member fields that have matching pointcuts in a collection, this collection would then be matched against at *every* pointer access. Since the number of pointcuts on fields is typically low the collection would be relatively small. The problem is that the checks have to be performed so many times. The points-to analysis reduce the number of places, as well as the size of the collection, to only those values that the pointer can actually point to. Since it is possible to make do without a points-to analysis it is clear that the analysis does not have to be perfect. Instead it can be used to decrease the number of values checked to some degree.

In this chapter a points-to algorithm is described with the aim of making a set and get implementation as effective as possible. It is not a complete algorithm and does not bring anything new to the field of pointer analysis. It is an attempt to describe how pointer analysis could be used to solve the issues of aliasing faced when creating set and get joinpoints and the problems that arise when trying to apply the analysis to this problem. The algorithm is designed to be as simple and straightforward as possible, it should be easy to implement and understand. Later, if the technique is proven usefull, a more sophisticated algorithm can be used to give similar results but at more precision or more likely at greater speed. Just as with an optimizing compiler the level of precision could be an option for the user. This way developer-builds can be fast and imprecise while release-builds would trade

a long compile-time for greater precision.

The notation used for points-to relationships emphasizes which fields a pointer can point to instead of listing all possible pairs. For example `*pointer={a, b}` means that `pointer` has a maybe-alias relationship with field `a` and `b` while `*pointer={a}` means that `pointer` must alias `a`.

A points-to analysis has a few basic properties that describes how it works and what kind of precision can be expected of the output. Algorithms that are *flow-insensitive* do not take the control-flow information into account when performing the analysis. Instead a summary is produced giving only information of the relationships that exist after each method or the whole program has been executed. *Flow-sensitive* algorithms on the other hand give a solution at every statement in the program, with more precision as a result. Another property is *context-sensitivity*, this property shows if the algorithm keeps track of different calling-contexts or if a function call at one point in the program can introduce a points-to relationship at a call to the same function at another place in the program. Again the sensitive approach is more precise than the insensitive.

Other properties are how objects that are allocated dynamically, *Heap modeling*, and how composite objects, *Aggregate modeling*, are modeled. [17]

Heap modeling and Aggregate modeling are not interesting in this setting because they are involved with values at a finer granularity than needed here, Section 5.5 describes that it is only member-fields that are tracked - not real values.

Two other properties to consider when constructing a points-to algorithm is whether it should handle only one function at a time, *intra-procedural*, or if function-calls are considered, *inter-procedural*. Inter-procedural analysis is much more complicated but gives a higher precision to the analysis. Algorithms used in current production systems are intra-procedural since they give a high benefit considering the fast execution time. Usually algorithms are classified by the technique used for finding points-to information or aliases as well as by the above properties, two common classes of analyses are type-based and flow-based. A type-based analysis uses some form of type-information to find points-to information, either traditional types that are used by the language being analyzed, or custom types constructed to model the memory of the program. Treating the points-to problem as a type problem makes type inference algorithms applicable to the points-to problem. Flow-based algorithms on the other hand, work with the control-flow of the program and consider what paths can lead up to a statement and what aliases are created when following this path. [7]

The algorithm described below is flow-sensitive, context-sensitive and requires the whole program to be analyzed at the same time. Normally a pointer analysis has the task of modelling an unbounded number of objects [17], due to the limitations in 5.3 and the fact that it is member fields and not memory locations that is tracked, the number of objects is bounded by

the number of fields in the classes of the program.

5.1 Study of Points-to Algorithms

A lot of work has been done on points-to algorithms, summaries of the field have been compiled [17] as well as guides to which one to choose for different purposes [18]. This section summarizes a number of algorithms that have been studied prior to the construction of the custom points-to algorithm.

Steensgaard has constructed a points-to algorithm with an almost linear time complexity. This analysis is inter-procedural, flow-insensitive as well as context-insensitive, but fast. The efficiency for large programs was also one of the design goals since many previous algorithms was lacking in this regard. The algorithm is type-based where the types model memory locations and not the conventional data-types, the finished set of types models the points-to relationships that can occur when the program runs. The algorithm consists of two steps where the first is to assign one type variable to each program variable, next the program is analyzed one statement at a time insuring that the typing holds, according to the algorithms type-rules, for each statement. If the typing does not hold the two types are joined. This way the further along the analysis goes the more refined is the typing environment as well as the points-to relationships. Steensgaard has also made an implementation that shows the efficiency of the algorithm in practice. [23]

One algorithm which has provided a lot of inspiration is the flow-sensitive, context-sensitive and inter-procedural analysis presented in “Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers” [14]. This algorithm approximates points-to information between different stack-locations and uses two different approaches for heap based objects (dynamic objects) and stack based. The algorithm builds up an invocation graph, which has a node for each calling of a function to give the context-sensitivity. Central to the analysis is a set of rules that describes for each statement which relationships are created, which are destroyed and which are made less precise. To simplify the rules and their application the target program is translated into a simpler set of statements than in the original program, also replacing complicated expressions with several simpler ones. Function pointers are handled by using the available points-to information to look up possible functions that a function pointer can refer to. [14]

Other algorithms have also been studied, for example one using a procedure call graph (PCG) to enable interprocedural analysis, but not context-sensitivity [12]. Another algorithm is a promising but complex analysis that is both flow-sensitive, context-sensitive and inter-procedural using a technique called partial transfer functions (PTF) to summarize a functions behaviour to be able to avoid re-analyzing a function at later calling-contexts

[26].

5.2 Goal

The goal of the points-to algorithm is to establish all possible alias relationships between fields and pointers in a program. These relationships can then be used to determine which pointcuts should be triggered.

```
class A {
public: int value;
};

class B {
public: int value;
};

...
int *p;
A one, two;
B three;

if (...)
    p = &one.value;
else if (...)
    p = &two.value;
else
    p = &three.value;

*p = 0;
...
```

Running the points-to algorithm on the above code gives that at `*p = 0;` the pointer `p` can point to the value field in either one, two or three which in turn mean that `*p = 0` is a set on either `A::value` or `B::value`. Let the following code replace `*p = 0;` and it will trigger the appropriate pointcut.

```
...
if (p == &one.value || p == &two.value)
    // Trigger set on A::value
else if (p == &three.value)
    // Trigger set on B::value
...
```

The problem with the values the pointer is compared against (`&one.value` etc.) can be out of scope is described in Section 5.5.

5.3 Limitations

This is a summary of the limitations that helps to give a simpler analysis of the points-to problem.

The implementation is thought to be used in evaluating fault tolerance mechanisms for embedded systems, especially systems that are embedded in cars. Because of this context the restrictions on the target programs are based on a guideline used by the motor industry. The MISRA C guideline [9], or “Guidelines For The Use Of The C Language In Vehicle Based Software” is a collection of required and advisory rules that help to create more secure and maintainable code.

The following limitations have been used to simplify the points-to analysis:

1. Do not consider pointer arithmetic.
2. Do not allow recursive functions.

Limitation 1 might seem a bit abrupt. It is not possible to solve the alias problem completely without this limitation, if we would allow pointer arithmetic all we could hope for would be a good estimation of the locations that a pointer can point to. With pointer arithmetic out of the way a pointer can only point to a location that it has some relationship with already in the source code, either direct or by a dependency. In contrast, if we would allow the pointer to be calculated at runtime it could point to an arbitrary location. To remove pointer arithmetic is actually not a big problem, in fact in many situations it is advisable not to use it since it is error prone. The MISRA C guideline states in Rule 101 (advisory) that “Pointer arithmetic should not be used”. Also consider that set and get pointcuts can be set on member fields only and member fields are seldom accessed by using pointer arithmetic.

Recursive functions (limitation 2) is another obstacle to pointer analysis. The problem with recursion is that when evaluating a function every calling context should be considered, but for every time the function is evaluated a new calling context is found creating an unlimited number of calling contexts. Unlike a real execution the analysis has to follow every path of the program both recursive and non-recursive. Approximation of the call path to handle recursion in points-to analysis is possible [26]. Limitation 2 is also justified by Rule 70 in the MISRA C guideline [9] which requires that “Functions shall not call themselves, either directly or indirectly”.

5.4 Algorithm

The basic idea of the algorithm is to analyze the program statement by statement, letting each statement affect the state of the relationships be-

tween pointers and fields. This is accomplished by visiting each statement at least once. Statements inside a loop will be visited twice or more and statements inside a function will be visited once for each call to the function. Every statement can be thought of as a transfer function that takes a state of all the relationships and gives the state as it would be if the statement had been executed.

There are two types of statements interesting to the algorithm; assignments and control flow. Assignments will kill old relationships and form new ones - an assignment can not be optional on its own so it will always create must-alias relationships. Control flow statements combine the assignments so that pointers may alias one of many fields.

The simplest case is the ordinary assignment of a pointer to the address of a value. In C++ this, would typically look like this:

```
int* pointer = &object.field;
```

After executing this statement, `pointer` can only point to one place. Any previous relationships the pointer has been assigned up to the statement will be destroyed. This statement is illustrated in Figure 5.1.

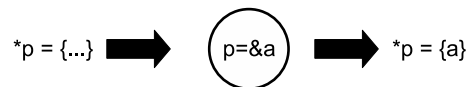


Figure 5.1: An assignment replaces any previous mappings for the pointer.

Another type of assignment is between two pointers where the assigned pointers are set to have the same relationships as the other pointer. The two pointers will not depend on each other after the statement, take this example where `p2` will keep pointing to `a` even after `p1` has been reassigned:

```
int* p1 = &a; // *p1 = {a}
int* p2;
p2 = p1; // *p2 = {a}
p1 = &b; // *p1 = {b}, *p2 = {a}
```



Figure 5.2: An assignment from one pointer to another will replace any previous mappings with a copy of the other pointer's mappings.

A single if-statement will treat its body as a single statement. The analysis should be applied to the body, and any pointers that are set when

the analysis finishes should be added to the current mappings. If-statements with several branches work similarly, in this case the analysis should be applied to all branches in parallel and then combined so that the pointers set in any or several of the branches are added to that pointer's possible mappings, see Figure 5.3 for an example. In case a pointer is set in all branches of the if-else or if-elseif-else any mappings that existed before the statement is replaced, this only applies if the last branch is an else-branch (In rule 60 MISRA advise that this should always be the case), this case can be seen in Figure 5.4.

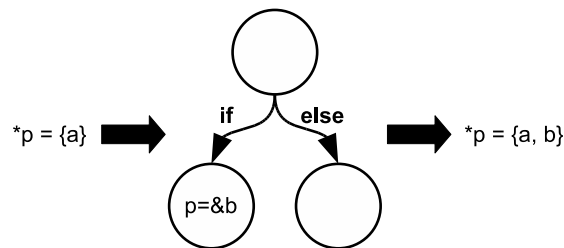


Figure 5.3: An if-else statement that does not set the pointer in all branches will add to the values a pointer can point to.

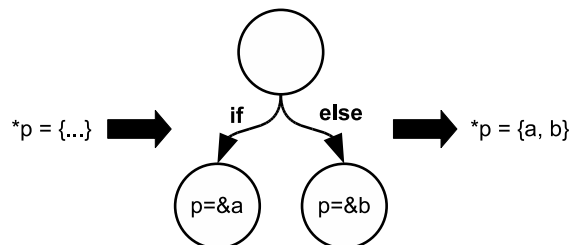


Figure 5.4: An if-else statement with an assignment of a pointer in all branches sets the pointer to one value per branch.

Loop-statements (**while** and **for** have the same behaviour and are considered equal in the analysis) also consider their bodies as a single statement that should be analyzed. In a real execution of the program the body of the loop will be executed zero or more times. Zero times means that mappings created in the body should be added to previous mappings and more times mean that the body can affect itself. To allow this self-affecting to take place, the analysis is run twice on the body of the loop. A simple example of a loop is shown in Figure 5.5. In Figure 5.6 it can be seen that values assigned to a pointer at the end of the loop affects which values the pointer maps to at the beginning.

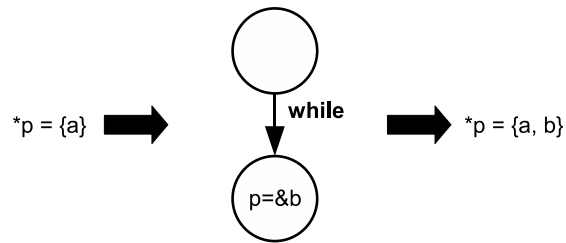


Figure 5.5: In its simplest form a loop will add to the mappings of a pointer.

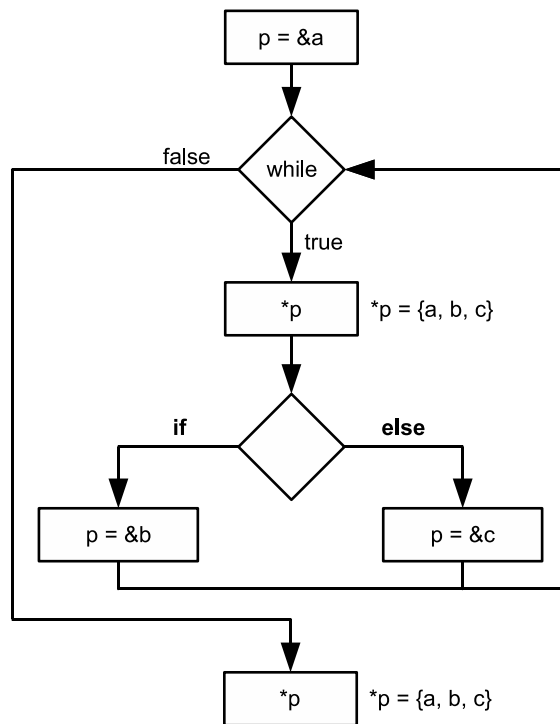


Figure 5.6: Loop where the latter part of the body affects the mappings at the beginning.

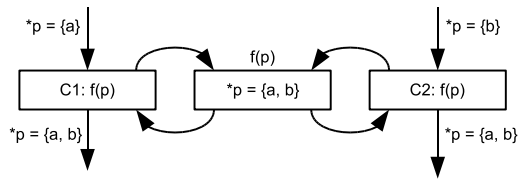


Figure 5.7: Two calling contexts contaminating each other.

Statements with substatements like `if` statements and loops will add one level of scope, meaning that any new relationships created in the body will only last after the statement if the affected pointers were in scope at the level of the `if` or loop statement.

This far only intraprocedural analysis has been discussed, that is the analysis has worked with a tree of statements that do not contain any function calls. Function calls provide a few challenges to pointer analysis: Recursion is one which has already been explained and avoided. Contamination of calling contexts is another problem which is not as severe as recursion but can lessen the precision of the analysis. Another problem is that it can become very complex and slow to process a large hierarchy of functions.

A function is generally called from many contexts and every time the function is called it may affect the possible states after the current calling context. For example, if a function `f` is called from two calling contexts; C1 and C2, and the pointer `p` points to the field `a` at C1 and `b` at C2 then inside the function `p` has a points-to relationship with both fields. Only one of these relationships, which one depends on the context, should remain after the function call. [26]

If `p` would have a relationship with both `a` and `b` after the call the calling-context is contaminated by a relationship which cannot occur. Contamination is illustrated in Figure 5.7.

In the points-to analysis the aliases both inside a function and at every calling context is of interest, hence to avoid contamination the relationships should be separated by calling context as shown in Figure 5.8. By separating the calling context like this the analysis becomes context-sensitive.

Function pointers will also need special attention, they work mostly in the same way as functions with the difference that it is not known when the analysis is run which function will be called at run-time. The solution is to run the analysis on all functions that can be invoked at the call-site and let their combined solution be the result of the function call. Without any information on which functions can be invoked from a given call-site all functions in the program, or maybe only those with a matching type-signature, would have to be considered and analyzed. But since this is a pointer analysis there

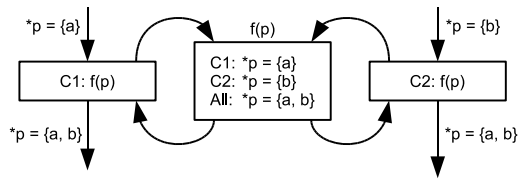


Figure 5.8: Relationships separated by calling context.

is information about possible functions that a function-pointer can point to available. [14]

There is a potential problem when a function returns a function pointer, if such a function is called with a function pointer from inside a loop, and the returned value is assigned to the function pointer that was used for calling. This situation implies that every iteration will call a different function from the last. This in turn implicates that it is not enough with two iterations for a loop, instead the loop should be analyzed until the points-to relationships doesn't change from one iteration to the next.

When several levels of indirection (pointers are pointing to other pointers) is used the analysis can utilize the current state of points-to information to follow the pointer and decide which value is actually affected. As an example, let the pointer p point to either one of two other pointers, $*p = \{p1, p2\}$. These pointers will also have relationships, lets define them as $*p1 = \{a, b\}$ and $*p2 = \{c\}$. Which value will $**p$ correspond to now? The answer is found by following the relationships resulting in $**p = \{a, b, c\}$. Assigning values to an indirected pointer will add new points-to relationships for all found values, if $*p = d$ is executed in the above state both pointers which $*p$ can point to will have a new relationship with d created giving $*p1 = \{a, b, d\}$ and $*p2 = \{c, d\}$. If a pointer has a definite points-to relationship the new relationship will not be added to but instead replace the old relationships. A simple example of this can be seen by redefining the pointers $*p = \{p1\}$ and $*p1 = \{a\}$, if now $**p = b$ is executed the relationship $*p1 = \{a\}$ will be replaced with $*p1 = \{b\}$. The same principles can be used on any number of levels of indirection, however the MISRA guideline only allows for two levels.

5.5 Invisible Variables

When entering a function the scope of the calling context is unavailable. This means that it is not possible to access the variables that are available at the caller by name, the variables are said to be invisible [14]. This raises the question on how to represent alias relationships with function calls present.

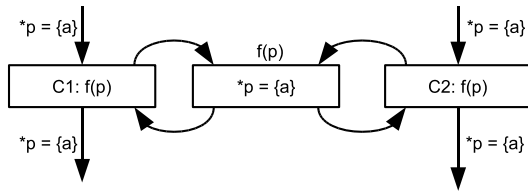


Figure 5.9: Name clash between two different calling-contexts.

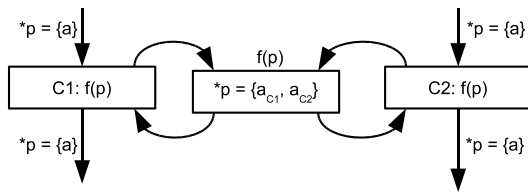


Figure 5.10: Labels used to differentiate between calling-contexts.

Another problem is name clashes, how is it possible to separate between variables with the same name from different calling contexts.

Invisible variables is a problem when weaving advice code. As seen in 5.2 the wanted structure of the weaved code needs to compare against the variables a pointer may point to, but if these variables are invisible it is not possible to perform the comparisons.

One way of distinguishing variables is to label them by their calling context [26]. With labels it is possible to identify which values can occur at which calling-context. This information can be used to avoid contamination. However, to be able to know when to trigger a pointcut all possible pointer values are needed at runtime. So, if the name of these values are out of scope, labels make no difference.

To get access to variables inside a function the function parameters could be extended to allow the variables to be passed along as well. This would put the names in scope for the function, but would lead to very long parameter lists which is not very efficient since all parameters are pushed onto the stack before a function call. The function structure is also destroyed making it impossible to link the program against object files not affected by the aspect compiler.

What is needed is actually not access to the variables that are out of scope, instead it is only information about which class-member a pointer points to at a certain time. Instead of determining this information by comparing it with the location of the original value it would be much more preferable to be able to get this information with access to the pointer only.

This could be accomplished in different ways. For example by mapping memory locations to an identifier, where every identifier represents a class-member. Such a map would need to be filled up with information while the program is running and every time a new object is created all its members should be mapped as members of that object's class. Code to handle this could be introduced to all classes constructors.

Another way to achieve this mapping would be to incorporate the identifiers into values giving an identification of which member and of which class a given value is. Practically the incorporation could be implemented with an extra member variable introduced into all classes. This member would contain the identification number of the field that particular object is stored in. To initialize this number, functionality would be added to all classes constructors where every member field has its identification number set. The following example shows how the identification numbers work.

```
// All fields have their own identification
// so these values would be global
enum {A_VALUE, B_VALUE};

class Value {
    int id; // Introduced identification
};

class A {
public:
    Value value;
    A();
};

class B {
public:
    Value value;
    B();
};

A::A() {
    value.id = A_VALUE;
}

B::B() {
    value.id = B_VALUE;
}
```

Both the map and the identification would allow for basically constant time lookup of which member-field a pointer is pointing to. The map would

probably require a bit more memory since the data structure would likely have some overhead. For both solutions some action is needed every time a new object is created, the map needs to be updated and identification numbers needs to be set. The major drawback of introducing an identification number is that it can only be done on classes leaving the primitive types unhandled.

These solutions will solve the invisible variable problem making it feasible to select which pointcuts to trigger with access to the pointer only and not the original values. In Section 5.2 the following snippet was used to select the right pointcut to trigger:

```
...
if (p == &one.value || p == &two.value)
    // Trigger set on A::value
else if (p == &three.value)
    // Trigger set on B::value
...
```

This snippet would not be possible to use because the `one`, `two` and `three` variables can be invisible. Instead this would be replaced with code to lookup which class `*pointer` is a member of. Either by looking it up in some form of global map:

```
...
if (memberMap.get(p) == A_VALUE)
    // Trigger set on A::value
else if (memberMap.get(p) == B_VALUE)
    // Trigger set on B::value
...
```

or by following the pointer and checking its identification:

```
...
if (p->id == A_VALUE)
    // Trigger set on A::value
else if (p->id == B_VALUE)
    // Trigger set on B::value
...
```

One noteworthy difference between these new solutions and the one where the pointer is compared against the address of variables is that in the new case only one comparison is needed for every member field instead of one comparison per object. This difference is important since it allows the pointer analysis to keep track of which fields pointers can point to instead of naming the instances. For the example above the previously used notation for the pointer `p` would be `p={one.value, two.value,`

`three.value`}, where `a` and `b` are variables, this notation can be substituted with `pointer={A::value, B::value}`, where `A::value` and `B::value` are fields, this substituted notation reflects that only two comparisons are needed instead of three.

5.6 Effectivity

A number of problems exists regarding the effectivity of the algorithm.

Having to run the analysis on the whole program at every compilation is a big obstacle. Normally only the changed source files need to be parsed at a compilation but if the analysis needs the whole program all files need to be parsed every time. This can take a substantial amount of time for large programs making it unfeasable to use the algorithm. It is not necessary to run the whole compilation process on every file; it is only the parsing and pointer analysis which needs to be run. The full compilation is only needed for modified files and files changed by the analysis.

The same code can be analyzed a large number of times because of function calls and loops. If there is a function call inside a loop the function will be analyzed twice, but if the loop is inside another loop the number of times double to four times. For large programs this kind of growing complexity can be devastating.

As stated in Section 3.1.2 the practice of taking the address of a member field is relatively rare in software, this is not a fact but can still be used to optimize the algorithm. By performing the analysis only when there is pointers to member fields present, and in other areas of the program only scan for the presence of them, it should be possible to speed up the analysis considerably. Research has been performed [16] to evaluate if it is possible to speed up a points-to analysis by only calculating the absolutely necessary values to be able to solve a limited part of the analysis. This research shows that a large overhead is introduced to perform the additional functionality involved, but if the points-to relationships sought are few and there isn't too many dependencies involved the demand-driven approach is very beneficial.

Chapter 6

Conclusions

In this thesis the problematics of implementing set and get for a language with pointers have been analyzed. Specifically C++ has been examined. The main problem when pointers are present is how to decide which value is actually accessed when using a pointer. Another problem with C++ is the ambiguity that occurs when a custom assignment operator is defined. Such custom operations does not necessarily need to perform assignments.

An extension has been implemented for the AspectC++ compiler that provides functionality to place pointcuts on the reading and writing of member fields. The joinpoints these pointcuts can match is limited to named access, so the pointer problems are disregarded. The extended compiler can be used for experimental evaluation of different aspect-mechanisms that require the set and get pointcuts. When working in an experimental environment it is not crucial that all features are available since heavy restrictions can be put on the programs that are compiled.

A possible solution to the problems concerning pointers is discussed in the form of a points-to analysis. By using this kind of analysis the number of checks needed to decide which field a pointer is pointing to can be decreased, in turn allowing for better performance of the compiled program. The work on this approach includes a study of a few different points-to algorithms, which problems are specific to this application of points-to analysis as well as an algorithm that would give a suitable output to use for the problem at hand.

6.1 Extensions and Future Work

To be able to extend the compiler a way of identifying which field a pointer is pointing to is needed, two approaches are described in Section 5.5.

When an identification system is in place the next step towards a more general set and get should probably be to implement a check-everything version of the compiler to handle pointers using brute-force. This way, the

usefulness of being able to trigger on pointer-access can be evaluated as well as the worst case performance.

After a brute-force implementation, a simple points-to analysis could be devised to remove some of the checks. This could be a simple intra-procedural analysis that for example checks if a pointer can receive a value from outside the current scope. If that is the case, make a thorough check, otherwise only test against values that are assigned inside the function.

The optimization can go on adding layers of new functionality to the points-to analysis giving better and better performance for the target program while the compilation process becomes gradually slower. When a reasonable balance between these two performances is found an optimal solution is reached. At this stage several solutions with different properties exist and a way of selecting different kinds of builds would be an interesting notion; a developer-build could be fast but produce a slow program while a production-build would be slow but give a better result.

An alternative approach to the pointer problem could also be considered, instead of using points-to analysis a similar solution as the one for identifying fields in Section 5.5 could be used. The identification is based on embedding information in the object about which field it is stored in. This concept could instead be used to embed information on what aspect-code to invoke when the value is read or written. This information could be as simple as function pointers to before, after and around advice. This solution has a few problems in that different aspect code should be triggered depending on the context the reading or writing takes place. Another problem is the invasive nature of the solution where the structure of the data is modified to allow for extra information. The advantage would be a simpler compilation process than with the points-to analysis as well as a constant time overhead on every read and write operation when using pointers.

List of Figures

2.1	The number of publications found in the ACM Digital Library per year with “aspect-oriented programming” either in the title or as a keyword. Source: ACM Digital Library [1]	7
2.2	Implementation of a cross-cutting concern with a subroutine (a) and an aspect (b).	7
3.1	Simple form of set and get.	11
3.2	The alias problem.	12
3.3	The alias problem with a pointer.	12
3.4	Randomized assignment of pointer.	12
3.5	Referenced access in Java	16
3.6	Pointer access to a field in C++, this scenario can not occur in Java.	16
3.7	Copying of pointer in C++, this is the same situation as in Figure 3.5.	16
4.1	Basic flow of operations in the AspectC++ compiler	18
4.2	Much simplified UML Class Diagram of the AspectC++ compiler with a few set and get specific classes added.	22
5.1	An assignment replaces any previous mappings for the pointer.	28
5.2	An assignment from one pointer to another will replace any previous mappings with a copy of the other pointer’s mappings.	28
5.3	An if-else statement that does not set the pointer in all branches will add to the values a pointer can point to.	29
5.4	An if-else statement with an assignment of a pointer in all branches sets the pointer to one value per branch.	29
5.5	In its simplest form a loop will add to the mappings of a pointer.	30
5.6	Loop where the latter part of the body affects the mappings at the beginning.	30
5.7	Two calling contexts contaminating each other.	31
5.8	Relationships separated by calling context.	32
5.9	Name clash between two different calling-contexts.	33
5.10	Labels used to differentiate between calling-contexts.	33

Appendix A

AspectC++ Source Code Modifications

Source code modifications done on the source-tree `ac-0.9.3/AspectC++`, the listing was created using the GNU diff program [8].

```
Only in from/AspectC++: .project
diff -r from/AspectC++/ACConfig.cc to/AspectC++/ACConfig.cc
78d77
<
diff -r from/AspectC++/ACWeaver.cc to/AspectC++/ACWeaver.cc
80a81,86
>
> // Johan: Set and Get added
> _pointcut_defs << "pointcut get(...) = 1;" << endl;
> _pointcut_defs << "pointcut set(...) = 1;" << endl;
> // --
>
104c110,114
< _pointcut_defs << " Target *target();" << endl;
---
> _pointcut_defs << " Target *target();" << endl;
> // Johan
> _pointcut_defs << " Result *source();" << endl;
> _pointcut_defs << " Result *dest();" << endl;
> // --
diff -r from/AspectC++/AspectIncludes.h to/AspectC++/AspectIncludes.h
50,51c50,51
<
< class AspectIncludes : private UnitSetMap {
---
> // Johan: made public inheritance --
> class AspectIncludes : public UnitSetMap {
diff -r from/AspectC++/Binding.cc to/AspectC++/Binding.cc
26c26,30
< _result = 0;
---
> _result = 0;
> //Johan
> _source = 0;
> _dest = 0;
> //--
33c37,42
```

```

<     _args.length () != other._args.length () || _result != other._result)
---
>     _args.length () != other._args.length () || _result != other._result
>     //Johan
>     || _source != other._source
>     || _dest != other._dest
>     // --
>     )
49c58,64
<     return BIND_RESULT;
---
>     return BIND_RESULT;
>     //Johan
>     else if (arg == _source)
>         return BIND_SOURCE;
>     else if (arg == _dest)
>         return BIND_DEST;
>     //--
diff -r from/AspectC++/Binding.h to/AspectC++/Binding.h
31,32c31,36
<     enum { BIND_THAT = -1, BIND_TARGET = -2, BIND_RESULT = -3,
<           BIND_NOT_FOUND = -4 };
---
>     enum { BIND_THAT = -1, BIND_TARGET = -2, BIND_RESULT = -3,
>           //Johan
>           BIND_SOURCE = -4,
>           BIND_DEST = -5,
>           //--
>           BIND_NOT_FOUND = -6 };
37c41,45
< CArgumentInfo *_result;
---
> CArgumentInfo *_result;
> //Johan
> CArgumentInfo *_source;
> CArgumentInfo *_dest;
> // --
diff -r from/AspectC++/CodeWeaver.cc to/AspectC++/CodeWeaver.cc
45a46,50
> //Johan
> #include <iostream>
> using namespace std;
> // --
>
85a91
>
140c146,150
< acdef << "    void *_that;" << endl;
---
> acdef << "    void *_that;" << endl;
> // Johan
> acdef << "    void *_source;" << endl;
> acdef << "    void *_dest;" << endl;
> //--
563c573,622
< JPL_Code *loc = (JPL_Code*)badly_typed_loc;
---
> JPL_Code *loc = (JPL_Code*)badly_typed_loc;
>
> //Johan: proceed code for intercepted set joinpoint
>
> // Get the result type

```

```

> JPL_Type jpl_result_type = loc->result_type ();
> CTypeInfo *dest_source_type = jpl_result_type.type_info();
>
> if (loc->type () == JoinPointLoc::Set) {
>   if (action) {
>     // Cast target to a useable type
>     out << " ";
>     dest_source_type->print(out);
>     out << "* dest = (";
>     dest_source_type->print(out);
>     out << "*)action._dest;" << endl;
>
>     // Cast that to a useable type
>     out << " ";
>     dest_source_type->print(out);
>     out << "* source = (";
>     dest_source_type->print(out);
>     out << "*)action._source;" << endl;
>
>     // Make the assignment
>     out << " *dest = *source;" << endl;
>
>     make_action_result_assignment (out, dest_source_type);
>     out << "*dest);" << endl;
>   }
>   else {
>     make_result_assignment (out, dest_source_type);
>     out << "*dest = source);" << endl;
>   }
>   return;
> }
> else if (loc->type () == JoinPointLoc::Get) {
>   if (action) {
>     make_action_result_assignment (out, dest_source_type);
>     out << "(";
>     dest_source_type->print(out);
>     out << "*)action._source);" << endl;
>   }
>   else {
>     make_result_assignment (out, dest_source_type);
>     out << "*source);" << endl;
>   }
> }
> // --
>
755a815
>
861c921,927
<         break;
---
>         break;
>         // Johan: Target is the variable that is being set when assigning
>         case JoinPointLoc::Get:
>         case JoinPointLoc::Set:
>             init << " target";
>             break;
>         // --
864,865c930,932
<     }
<     } else {
---
>     }

```

```

>     }
>     else {
889c956,962
<         break;
---
>         break;
>         // Johan:
>         case JoinPointLoc::Get:
>         case JoinPointLoc::Set:
>             init << " that";
>             break;
>             // --
892,893c965,967
<         }
<     } else {
---
>     }
>     }
>     else {
896c970,998
<     }
---
>     }
>
>     //Johan
>     // Pass along the source object, only set and get has this, all others get 0
>     if (tjp.source() || tjp.useAction()) {
>         if (fields) init << ", ";
>         fields++;
>         if (jptype == JoinPointLoc::Set) {
>             init << " &source";
>         }
>         else if (jptype == JoinPointLoc::Get) {
>             init << " source";
>         }
>         else {
>             init << "0";
>         }
>     }
>     // Only set has a destination
>     if (tjp.dest() || tjp.useAction()) {
>         if (fields) init << ", ";
>         fields++;
>         if (jptype == JoinPointLoc::Set) {
>             init << " dest";
>         }
>         else {
>             init << "0";
>         }
>     }
>     //--
999,1000c1101,1292
<
<
---
>
> // Johan: Implementation of set joinpoint invocation weaving
> void CodeWeaver::set_join_point (JPL_Set *loc, JPP_Code &plan) {
>
>     // Get the advice
>     JPAdvice *advice = plan.jp_advice();
>

```

```

> // Does this join point need an action?
> bool action = advice->tjp ().action();
>
> // Generate the call
> // Stream to hold the wrap function invocation
> ostream wrap_invocation;
> // Name
> Naming::set_wrap (wrap_invocation, loc);
> // Arguments
> wrap_invocation << "(";
>
> CT_BinaryExpr *assignment = (CT_BinaryExpr*)loc->tree();
> CTree *assignment_op = assignment->Son(1);
> CTree *left = assignment->Son(0);
>
> // The 'Target' pointer
> bool isMembPtr = left->NodeName () == CT_MembPtrExpr::NodeId();
> bool isMembRef = left->NodeName () == CT_MembRefExpr::NodeId();
> if (isMembPtr || isMembRef) {
>     CTree *object = left->Son(0);
>
>     // If it's a member reference take the address
>     if (isMembRef) {
>         wrap_invocation << "&";
>     }
>     print_tree(wrap_invocation, object);
>     wrap_invocation << ", ";
> }
> else {
>     wrap_invocation << "this, ";
> }
>
> // The 'That' pointer
> // Are we inside a class?
> if (loc->containing_func ()->Record ()) {
>     // 'that' pointer is 'this'
>     wrap_invocation << "this, ";
> }
> else {
>     // Not inside a class, so 'that' pointer is zero
>     wrap_invocation << "0, ";
> }
>
> // Pass the variable to be assigned by reference
> wrap_invocation << "&";
> // Paste the invocation
> paste (weave_pos (assignment->token (), WeavePos::WP_BEFORE),
>        wrap_invocation.str());
>
> // Replace the assignment operator (=) with an argument separator (,)
> replace(assignment_op, "=", ",");
> // Put in the closing ) to end the call
> paste (weave_pos (assignment->end_token(), WeavePos::WP_AFTER), ")");
> // End of call generation --
>
> // Generate the wrap definition
> // Stream to hold the wrap function definition
> ostream wrap_definition;
>
> // Snuck in the action wrapper before the "actual" wrapper if needed
> if (action) {
>     make_action_wrapper(wrap_definition, loc, advice);

```

```

>     wrap_definition << endl;
> }
>
> // Get the result type once and for all
> JPL_Type jpl_result_type = loc->result_type();
> CTypeInfo *result_type = jpl_result_type.type_info();
>
> // Make it inline
> wrap_definition << "inline ";
> // Type
> result_type->print(wrap_definition);
> wrap_definition << " ";
> // Name
> Naming::set_wrap (wrap_definition, loc);
> // Arguments
> wrap_definition << "(";
> loc->target_type()->print(wrap_definition);
> wrap_definition << " *target, ";
> loc->that_type()->print(wrap_definition);
> wrap_definition << " *that, ";
> result_type->print(wrap_definition);
> wrap_definition << " *dest, ";
> result_type->print(wrap_definition);
> wrap_definition << " source) {" << endl;
> // Body
> // Result buffer
> make_result_declaration(wrap_definition, result_type);
> // generate common JoinPoint initialization
> make_tjp_common_init(wrap_definition, loc, advice);
> // Generate calls to advice
> make_advice_calls(wrap_definition, advice, loc);
> // Return result
> make_result_return (wrap_definition, result_type);
> wrap_definition << "}" << endl;
>
> // Paste the definition right after the tjp struct
> paste (weave_pos (loc->insertloc_wrap(), WeavePos::WP_BEFORE),
>     wrap_definition.str());
>
> }
> //--
>
> // Johan: Implementation of get joinpoint weaving
> void CodeWeaver::get_join_point (JPL_Get *loc, JPP_Code &plan) {
>
>     // Get the advice
>     JPAdvice *advice = plan.jp_advice();
>
>     // Does this join point need an action?
>     bool action = advice->tjp ().action();
>
>     // Generate the call
>     // Stream to hold the wrap function invocation
>     ostream wrap_invocation;
>     // Name
>     Naming::get_wrap (wrap_invocation, loc);
>     // Arguments
>     wrap_invocation << "(";
>
>     bool isMembPtr = loc->tree()->NodeName () == CT_MembPtrExpr::NodeId();
>     bool isMembRef = loc->tree()->NodeName () == CT_MembRefExpr::NodeId();
>     if (isMembPtr || isMembRef) {

```

```

> CT_BinaryExpr *bin_expr = (CT_BinaryExpr*)loc->tree ();
> if (isMembRef) {
>   wrap_invocation << "&";
> }
> print_tree (wrap_invocation, bin_expr->Son(0));
> }
> else {
>   wrap_invocation << "this";
> }
> // Pass the variable to be assigned by reference
> wrap_invocation << ", &(";
> // Paste the invocation
> paste (weave_pos (loc->tree ()->token (), WeavePos::WP_BEFORE),
>   wrap_invocation.str());
>
> // Put in the closing ) to end the call
> paste (weave_pos (loc->tree ()->end_token(), WeavePos::WP_AFTER, ")");
> // End of call generation --
>
> // Generate the wrap definition
> // Stream to hold the wrap function definition
> ostreamstream wrap_definition;
>
> // Snuck in the action wrapper before the "actual" wrapper if needed
> if (action) {
>   make_action_wrapper(wrap_definition, loc, advice);
>   wrap_definition << endl;
> }
>
> // Make it inline
> wrap_definition << "inline ";
> // Type
> loc->target_type()->print(wrap_definition);
> wrap_definition << " ";
> // Name
> Naming::get_wrap (wrap_definition, loc);
> // Arguments
> wrap_definition << "(";
> loc->that_type()->print(wrap_definition);
> wrap_definition << " *that, ";
> loc->target_type()->print(wrap_definition);
> wrap_definition << " *source) {" << endl;
> // Body
> // Result buffer
> make_result_declaration(wrap_definition, loc->target_type());
> // generate common JoinPoint initialization
> make_tjp_common_init(wrap_definition, loc, advice);
> // Generate calls to advice
> make_advice_calls(wrap_definition, advice, loc);
> // Return result
> make_result_return (wrap_definition, loc->target_type());
> wrap_definition << "}" << endl;
>
> // Paste the definition right after the tjp struct
> paste (weave_pos (loc->insertloc_wrap(), WeavePos::WP_BEFORE),
>   wrap_definition.str());
>
> }
> //--
>
>
1010c1302

```

```

<
---
>
1012c1304
<
---
>
1419c1711,1719
< }
---
> }
> // Johan: Handle set and get join point type
> else if (loc->type () == JoinPointLoc::Set) {
> // Do nothing
> }
> else if (loc->type () == JoinPointLoc::Get) {
> // Do nothing
> }
> // --
1685c1985,1986
<
---
>
> // Johan: Don't paste a new parenthesis, let the old one close the call
1687,1688c1988,1989
< paste (weave_pos (loc->CallExprNode ()->end_token (), WeavePos::WP_AFTER),
< " ");
---
> //paste (weave_pos (loc->CallExprNode ()->end_token (), WeavePos::WP_AFTER),
> // " ");
1737,1738c2038,2040
< kill (args->Son (0));
< kill (args->Son (args->Sons () - 1));
---
> kill (args->Son (0));
> //Johan: Don't remove the parenthesis, let it form the ending of the call
> //kill (args->Son (args->Sons () - 1));
diff -r from/AspectC++/CodeWeaver.h to/AspectC++/CodeWeaver.h
118a119,122
> //Johan: Weave invocation code for a set joinpoint
> void set_join_point (JPL_Set *loc, JPP_Code &plan);
> void get_join_point (JPL_Get *loc, JPP_Code &plan);
> // --
diff -r from/AspectC++/JoinPointLoc.cc to/AspectC++/JoinPointLoc.cc
378a379,548
>
> //Johan: Set join point locations implementation
> JPL_Set::JPL_Set(
> CT_SimpleName *variable,
> Token *op,
> CTree *source,
> CTree *assignment,
> CFunctionInfo *parent,
> int local_id) {
>
> // Set members
> _variable = variable;
> _op = op;
> _source = source;
> _parent = parent;
> _lid = local_id;
> _assignment = assignment;

```

```

>
> // Stream to create the signature with
> stringstream sig_stream;
> // Make the signature
> // Type
> _variable->Object ()->TypeInfo ()->print (sig_stream);
> sig_stream << " ";
> // Qualified name
> sig_stream << _variable->Object ( )->AttributeInfo ()->QualName ();
> // Set the return type of the JPL_Code base object to be of the same
> // type as the variable
> result_type(_variable->Type ());
> // Set the signature
> _sig = sig_stream.str ();
> }
>
> CTypeInfo *JPL_Set::target_type() {
> // Target should have the same type as the containing class
> CRecord *record = _variable->Object ()->AttributeInfo ()->Record ();
> CTypeInfo *type_info = record->TypeInfo ();
> return type_info;
> }
>
> CTypeInfo *JPL_Set::that_type() {
>
> // That has the type of the containing functions class
> CRecord *record = _parent->Record ();
> if (record) {
> return record->TypeInfo ();
> }
> return &CTYPE_VOID;
> }
>
> CObjectInfo *JPL_Set::assoc_obj() {
> return _variable->Object ();
> }
>
> // Where should the ThisJoinPoint structure be placed?
> Token *JPL_Set::insertloc_tjp_struct() {
>
> //CRecord *record = _variable->Object()->AttributeInfo()->Record();
> //return record->Tree ()->end_token ();
>
> // The tjp struct should be inserted before the function in which the
> // assignment takes place
> return _parent->Tree()->token();
> }
>
> // Where should the wrap function for the assignment of member
> // variables be
> Token *JPL_Set::insertloc_wrap () {
> // Get the record of the defining class
> //CRecord *record = _variable->Object ()->AttributeInfo ()->Record ();
> // Class info of defining class
> //CClassInfo *class_info = record->ClassInfo ();
> // Class definition tree of defining class
> //CT_ClassDef *class_def = (CT_ClassDef*)class_info->Tree ();
> // Last token of the members
> //Token *inspos = class_def->Members ()->end_token ();
> //return inspos;
> return insertloc_tjp_struct();
> }

```

```

>
> // JoinPointLoc for get
> JPL_Get::JPL_Get(CT_SimpleName *variable,
>   CTree *tree,
>   CFunctionInfo *parent,
>   int local_id) {
>
>   // Set members
>   _variable = variable;
>   _lid = local_id;
>   _tree = tree;
>   _parent = parent;
>
>   // Stream to create the signature with
>   stringstream sig_stream;
>   // Make the signature
>   // Type
>   _variable->Object ()->TypeInfo ()->print (sig_stream);
>   sig_stream << " ";
>   // Qualified name
>   sig_stream << _variable->Object ()->AttributeInfo ()->QualName ();
>   // Set the return type of the JPL_Code base object to be of the same
>   // type as the variable
>   result_type(_variable->Type ());
>   // Set the signature
>   _sig = sig_stream.str ();
> }
>
> CTypeInfo *JPL_Get::target_type() {
>   // Target type is the same as the result type
>   JPL_Type jpl_type = result_type ();
>   CTypeInfo *type_info = jpl_type.type_info ();
>   return type_info;
> }
>
> CTypeInfo *JPL_Get::that_type() {
>   // That should have the same type as the containing class
>   CRecord *record = _variable->Object ()->AttributeInfo ()->Record ();
>   CTypeInfo *type_info = record->TypeInfo ();
>   return type_info;
> }
>
> CObjectInfo *JPL_Get::assoc_obj() {
>   return _variable->Object ();
> }
>
> // Where should the ThisJoinPoint structure be placed?
> Token *JPL_Get::insertloc_tjp_struct() {
>
>   //CRecord *record = _variable->Object()->AttributeInfo()->Record();
>   //return record->Tree ()->end_token ();
>
>   // The tjp struct should be inserted before the function in which the
>   // assignment takes place
>   return _parent->Tree()->token();
> }
>
> // Where should the wrap function for the assignment of memeber
> // variables be
> Token *JPL_Get::insertloc_wrap () {
>   return insertloc_tjp_struct();
> }
>

```

```

> // --
380,381c550
<
< JPL_FieldReference::JPL_FieldReference (CAttributeInfo *a,
----
> /*JPL_FieldReference::JPL_FieldReference (CAttributeInfo *a,
403c572
< }
----
> */
406c575
< JPL_FieldAssignment::JPL_FieldAssignment (CAttributeInfo *a,
----
> /*JPL_FieldAssignment::JPL_FieldAssignment (CAttributeInfo *a,
429,430c598
< }
<
----
> */
464d631
<
diff -r from/AspectC++/JoinPointLoc.h to/AspectC++/JoinPointLoc.h
57,59c57,61
<     None           = 0x0000,
<     FieldReference = 0x0001,
<     FieldAssignment = 0x0002,
----
>     None           = 0x0000,
>     Get            = 0x0001,
>     Set            = 0x0002,
>     //FieldReference = 0x0001,
>     //FieldAssignment = 0x0002,
279c281
< class JPL_FieldReference : public JPL_Code
----
> /*class JPL_FieldReference : public JPL_Code
292c294
< };
----
> */
294c296
< class JPL_FieldAssignment : public JPL_Code
----
> /*class JPL_FieldAssignment : public JPL_Code
309c311,359
< };
----
> */
>
> //Johan: Join point location for a 'set' on a variable
> class JPL_Set : public JPL_Code
> {
> private:
>     CT_SimpleName *_variable;
>     Token *_op;
>     CTree *_source;
>     CTree *_assignment;
>     CFunctionInfo *_parent;
> public:
>     JPL_Set (CT_SimpleName *variable, Token *op, CTree *source,
>             CTree *assignment, CFunctionInfo *parent, int local_id);
>     join_point_type type () { return Set; }

```

```

> virtual CTree *tree () const { return _assignment; }
> virtual const char *type_str () const { return "set"; }
> virtual CFunctionInfo *containing_func () { return _parent; }
> virtual CObjectInfo *assoc_obj ();
> virtual CTypeInfo *that_type ();
> virtual CTypeInfo *target_type ();
> virtual Token *insertloc_tjp_struct();
> Token *insertloc_wrap ();
> };
>
> class JPL_Get : public JPL_Code
> {
> private:
>   CT_SimpleName *_variable;
>   CFunctionInfo *_parent;
>   CTree *_tree;
> public:
>   JPL_Get (CT_SimpleName *variable, CTree *tree, CFunctionInfo *parent,
>     int local_id);
>   join_point_type type () { return Get; }
>   virtual CTree *tree () const { return _tree; }
>   virtual const char *type_str () const { return "get"; }
>   virtual CFunctionInfo *containing_func () { return _parent; }
>   virtual CObjectInfo *assoc_obj ();
>   virtual CTypeInfo *that_type ();
>   virtual CTypeInfo *target_type ();
>   virtual Token *insertloc_tjp_struct();
>   Token *insertloc_wrap ();
> };
> // --
diff -r from/AspectC++/JoinPointLocList.h to/AspectC++/JoinPointLocList.h
37c42,43
< enum { METHODS = 0, CALLS, CONS, DESTS, CLASSES, ASPECTS, FUNCS, NO };
---
> enum { METHODS = 0, CALLS, CONS, DESTS, CLASSES, ASPECTS, FUNCS,
>   SET, GET, NO };
83c89
< void new_field_reference (CAttributeInfo *a, CTree *f, int local_id) {
---
> /*void new_field_reference (CAttributeInfo *a, CTree *f, int local_id) {
85c91
< }
---
> }*/
87c93
< void new_field_assignment (CAttributeInfo *a, CT_CallExpr *as,
---
> /*void new_field_assignment (CAttributeInfo *a, CT_CallExpr *as,
89,90c95,122
< //   _field_assignments.push_back (new JPL_FieldAssignment (a, as, f, local_id));
< }
---
> }*/
>
> // Johan
> void new_set (CT_SimpleName *variable, Token *op, CTree *source,
>   CTree *assignment, CFunctionInfo *parent, int local_id) {
>
>   _elements[SET].push_back(new JPL_Set(
>     variable,
>     op,
>     source,

```

```

>     assignment,
>     parent,
>     local_id));
> }
>
> void new_get (CT_SimpleName *variable, CTree *tree,
>     CFunctionInfo *parent, int local_id) {
>
>     _elements[GET].push_back(new JPL_Get(variable, tree, parent, local_id));
> }
> // --
diff -r from/AspectC++/Naming.cc to/AspectC++/Naming.cc
270c270,274
< case JoinPointLoc::MethodCall: out << "call_"; break;
---
> case JoinPointLoc::MethodCall: out << "call_"; break;
> //Johan: action wrapper name for set and get join points
> case JoinPointLoc::Set: out << "set_"; break;
> case JoinPointLoc::Get: out << "get_"; break;
> // --
375a380,391
> // Johan: Implementation of the name generator for wraps around assignments
> void Naming::set_wrap(ostream& out, JoinPointLoc *loc) {
>     out << "__set_";
>     mangle(out, loc->assoc_obj());
>     if (loc->lid() >= 0) out << "_" << loc->lid();
> }
> void Naming::get_wrap(ostream& out, JoinPointLoc *loc) {
>     out << "__get_";
>     mangle(out, loc->assoc_obj());
>     if (loc->lid() >= 0) out << "_" << loc->lid();
> }
> //--
diff -r from/AspectC++/Naming.h to/AspectC++/Naming.h
70c70,74
< static void guard (ostream &out, FileUnit *unit);
---
> static void guard (ostream &out, FileUnit *unit);
> // Johan
> static void set_wrap(ostream& out, JoinPointLoc *loc);
> static void get_wrap(ostream& out, JoinPointLoc *loc);
> // --
diff -r from/AspectC++/Plan.cc to/AspectC++/Plan.cc
44c44,51
<     delete _dest_jpls[i]->plan ();
---
>     delete _dest_jpls[i]->plan ();
> // Johan: clean up set join point plans
> for (int i = 0; i < (int)_set_jpls.length (); i++)
>     delete _set_jpls[i]->plan ();
> for (int i = 0; i < (int)_get_jpls.length (); i++)
>     delete _get_jpls[i]->plan ();
>
> // --
89c96
<     jp_plan = new JPP_Code;
---
>     jp_plan = new JPP_Code();
93c100
<     jp_plan = new JPP_Code;
---
>     jp_plan = new JPP_Code();

```

```

97c104
<         jp_plan = new JPP_Code;
----
>         jp_plan = new JPP_Code();
101c108
<         jp_plan = new JPP_Code;
----
>         jp_plan = new JPP_Code();
103c110,120
<         break;
----
>         break;
>         //Johan:
>         case JoinPointLoc::Set:
>             jp_plan = new JPP_Code();
>             _set_jpls.append (jpl);
>             break;
>         case JoinPointLoc::Get:
>             jp_plan = new JPP_Code();
>             _get_jpls.append(jpl);
>             break;
>         // --
256c277,292
<     }
----
>     }
>     // Johan: Check set join point plans
>     for (int i = 0; i < (int)_set_jpls.length (); i++) {
>         if (!_set_jpls[i]->plan ()->check(_err)) {
>             _err << " for set(\"" << _set_jpls[i]->signature () << "\""
>             << _set_jpls[i]->tree ()->token ()->location () << endMessage;
>         }
>     }
>     // Check get join points
>     for (int i = 0; i < (int)_get_jpls.length (); i++) {
>         if (!_get_jpls[i]->plan ()->check(_err)) {
>             _err << " for set(\"" << _get_jpls[i]->signature () << "\""
>             << _get_jpls[i]->tree ()->token ()->location () << endMessage;
>         }
>     }
>     // --
diff -r from/AspectC++/Plan.h to/AspectC++/Plan.h
57c57,61
<     Array<JoinPointLoc*> _class_jpls;
----
>     Array<JoinPointLoc*> _class_jpls;
>     //Johan
>     Array<JoinPointLoc*> _set_jpls;
>     Array<JoinPointLoc*> _get_jpls;
>     // --
118a123,142
>
>     // Johan: get join point plan, location and total number of set join points
>     // in plan
>     int set_jp_plans () const { return _set_jpls.length (); }
>     JPL_Set &set_jp_loc (int i) const {
>         return *(JPL_Set*)_set_jpls.lookup (i);
>     }
>     JPP_Code &set_jp_plan (int i) const {
>         return *(JPP_Code*)set_jp_loc (i).plan ();
>     }
>

```

```

> int get_jp_plans () const { return _get_jpls.length (); }
> JPL_Get &get_jp_loc (int i) const {
>     return *(JPL_Get*)_get_jpls.lookup (i);
> }
> JPP_Code &get_jp_plan (int i) const {
>     return *(JPP_Code*)get_jp_loc (i).plan ();
> }
>
> // --
diff -r from/AspectC++/PointCutEvaluator.cc to/AspectC++/PointCutEvaluator.cc
183a184,194
>
>     // Johan: Create pointcut expressions for set
>     else if (strcmp (name, "set") == 0) {
>         // Create a point cut expression to handle the parameter to set
>         result = new PCE_Set (create (args->Entry(0)));
>     }
>     // Johan: Create pointcut expressions for get
>     else if (strcmp (name, "get") == 0) {
>         // Create a point cut expression to handle the parameter to get
>         result = new PCE_Get (create (args->Entry(0)));
>     }
diff -r from/AspectC++/PointCutExpr.cc to/AspectC++/PointCutExpr.cc
267c271,275
<     func = ((JPL_MethodCall&jpl).caller ()); break;
---
>     func = ((JPL_MethodCall&jpl).caller ()); break;
> //Johan: A set joinpoint loc has the same scope as it's containing function
> case JoinPointLoc::Set:
>     func = ((JPL_Set&jpl).containing_func ()); break;
> // --
302a311
>
343a353
>
353c363
<     if (class_loc && arg (0)->evaluate (*class_loc, context, binding, cond))
---
>     if (class_loc && arg (0)->evaluate (*class_loc, context, binding, cond)) {
354a365
>     }
359c370
<     if (func_loc && arg (0)->evaluate (*func_loc, context, binding, cond))
---
>     if (func_loc && arg (0)->evaluate (*func_loc, context, binding, cond)) {
360a372
>     }
412a425,483
> // Johan
> /**
>  * @return the type of the of the point cut expression
>  */
> PCE_Type PCE_Set::type () const {
>     return PCE_CODE;
> }
>
> /**
>  * TODO: find out what this function does
>  */
> void PCE_Set::semantics (ErrorStream &err,
>     PointCutContext &context) {
>     check_arg_types (err, "set", PCE_NAME);

```

```

> sem_args (err, context);
> }
>
> /**
> * Check if this point cut expression matches the given joinpoint location
> * @return true if the join point location matches this point cut expression,
> * otherwise false
> */
> bool PCE_Set::evaluate (JoinPointLoc &jpl, PointCutContext &context,
>                          Binding &binding, Condition &cond) {
>     // consider set join point only
>     if (jpl.type () != JoinPointLoc::Set) {
>         return false;
>     }
>     return arg(0)->evaluate(jpl, context, binding, cond);
> }
> // End PCE_Set
>
> // PCE_Get
> PCE_Type PCE_Get::type () const {
>     return PCE_CODE;
> }
>
> void PCE_Get::semantics (ErrorStream &err,
>                          PointCutContext &context) {
>     check_arg_types (err, "get", PCE_NAME);
>     sem_args (err, context);
> }
>
> /**
> * Check if this point cut expression matches the given joinpoint location
> * @return true if the join point location matches this point cut expression,
> * otherwise false
> */
> bool PCE_Get::evaluate (JoinPointLoc &jpl, PointCutContext &context,
>                          Binding &binding, Condition &cond) {
>     // consider get join point only
>     if (jpl.type () != JoinPointLoc::Get) {
>         return false;
>     }
>
>     return arg(0)->evaluate(jpl, context, binding, cond);
> }
> // End PCE_Get
> //--
917c988,1002
<  assert (!_match_expr.error ());
---
>  assert (!_match_expr.error ());
>
> //Johan
> //Add support for matching when the expression describes an attribute
> if (_match_expr.is_attribute()) {
>     if (jpl.type() == JoinPointLoc::Set ||
>         jpl.type() == JoinPointLoc::Get) {
>         JPL_Code &jpl_code = (JPL_Code&)jpl;
>         JPL_Type jpl_type = jpl_code.result_type ();
>         CTypeInfo *type = jpl_type.type_info ();
>         CObjectInfo *obj = jpl_code.assoc_obj();
>         return _match_expr.matches (type, obj);
>     }
> }

```

```

> // --
diff -r from/AspectC++/PointCutExpr.h to/AspectC++/PointCutExpr.h
246a247,275
> //Johan
> // Point cut expression for the set pointcut type
> class PCE_Set : public PCE_SingleArg {
> public:
>   PCE_Set (PointCutExpr *arg) : PCE_SingleArg (arg) {}
>   virtual PCE_Type type () const;
>   virtual void semantics (ErrorStream &err, PointCutContext &context);
>   virtual bool evaluate (JoinPointLoc &jpl, PointCutContext &context,
>                           Binding &binding, Condition &cond);
>   virtual void mangle_type_check (ostream &out) {
>     out << "set_";
>     PCE_SingleArg::mangle_type_check (out);
>   }
> };
> // Point cut expression for get pointcut
> class PCE_Get : public PCE_SingleArg {
> public:
>   PCE_Get (PointCutExpr *arg) : PCE_SingleArg (arg) {}
>   virtual PCE_Type type () const;
>   virtual void semantics (ErrorStream &err, PointCutContext &context);
>   virtual bool evaluate (JoinPointLoc &jpl, PointCutContext &context,
>                           Binding &binding, Condition &cond);
>   virtual void mangle_type_check (ostream &out) {
>     out << "get_";
>     PCE_SingleArg::mangle_type_check (out);
>   }
> };
> // --
>
diff -r from/AspectC++/ThisJoinPoint.cc to/AspectC++/ThisJoinPoint.cc
54c54,58
<   _used |= (binding._args.length () > 0) ? ARG : 0;
---
>   _used |= (binding._args.length () > 0) ? ARG : 0;
>   // Johan
>   _used |= (binding._source != 0) ? SOURCE : 0;
>   _used |= (binding._dest != 0) ? DEST : 0;
>   // --
143c147,153
<   _used |= (PROCEED|ACTION);
---
>   _used |= (PROCEED|ACTION);
>   // Johan
>   else if (strcmp (field, "source") == 0)
>     _used |= SOURCE;
>   else if (strcmp (field, "dest") == 0)
>     _used |= DEST;
>   // --
186,187c196,197
<   types << ";" << endl;
<
---
>   types << ";" << endl;
>
315c325,333
<   fct << "   inline That* that() {return (That*)_action->_that;}" << endl;
---
>   fct << "   inline That* that() {return (That*)_action->_that;}" << endl;
>   //Johan

```

```

>     if (source())
>         fct << " inline Result* source() {return (Result*)_action->_source;}"
>         << endl;
>     if (dest())
>         fct << " inline Result* dest() {return (Result*)_action->_dest;}"
>         << endl;
>         //--
340c358,368
<     }
---
>     }
>     // Johan
>     if (source()) {
>         data << " Result *_source;" << endl;
>         fct << " inline Result *source() {return _source;}" << endl;
>     }
>     if (dest()) {
>         data << " Result *_dest;" << endl;
>         fct << " inline Result *dest() {return _dest;}" << endl;
>     }
>     // --
346c374
<     fct << " return (typename Arg<I>::ReferredType*)arg (I);" << endl;
---
>     fct << " return (typename Arg<I>::ReferredType*)arg (I);" << endl;
370c398,400
<     out << ", that=" << (that() ? "true" : "false");
---
>     out << ", that=" << (that() ? "true" : "false");
>     out << ", source=" << (source() ? "true" : "false");
>     out << ", dest=" << (dest() ? "true" : "false");
diff -r from/AspectC++/ThisJoinPoint.h to/AspectC++/ThisJoinPoint.h
49c49,53
<     WRAPPER=0x1000, PROCEED=0x2000,
---
>     WRAPPER=0x1000, PROCEED=0x2000,
>     // Johan
>     SOURCE=0x4000,
>     DEST=0x8000,
>     //--
51c55
<     PTR_NEEDED=0x4000,
---
>     PTR_NEEDED=0x10000,
53c57
<     PTR_ALIAS_NEEDED=0x8000,
---
>     PTR_ALIAS_NEEDED=0x20000,
55c59
<     TYPE_NEEDED=0x10000
---
>     TYPE_NEEDED=0x40000
84c88,92
<     bool proceed() const { return (_used & PROCEED); }
---
>     bool proceed() const { return (_used & PROCEED); }
>     // Johan
>     bool source() const { return (_used & SOURCE); }
>     bool dest() const { return (_used & DEST); }
>     // --
diff -r from/AspectC++/TrackerDog.cc to/AspectC++/TrackerDog.cc
26a27,32

```

```

> //Johan
> #include "WeaverBase.h"
> // --
>
129,130c135
< void TrackerDog::pre_visit (CTree *node)
< {
---
> void TrackerDog::pre_visit (CTree *node) {
162c167
<     in_memb_init = true;;
---
>     in_memb_init = true;
170c175,196
< }
---
>
> //Johan: Check all binary expressions
> else if (id == CT_BinaryExpr::NodeId ()) {
>     pre_BinaryExpr((CT_BinaryExpr*)node);
> }
> else if (id == CT_SimpleName::NodeId()) {
>     pre_SimpleName((CT_SimpleName*)node);
> }
> else if (id == CT_MembRefExpr::NodeId ()) {
>     pre_MembRefPtrExpr((CT_BinaryExpr*)node);
> }
> else if (id == CT_MembPtrExpr::NodeId ()) {
>     pre_MembRefPtrExpr((CT_BinaryExpr*)node);
> }
> else if (id == CT_AddrExpr::NodeId ()) {
>     pre_AddrExpr((CT_AddrExpr*)node);
> }
> else if (id == CT_DerefExpr::NodeId ()) {
>     pre_DerefExpr((CT_DerefExpr*)node);
> }
> // --
> }
235,236c261,265
<     func_stack.push (func->FunctionInfo ());
<     local_id = 0;
---
>     func_stack.push (func->FunctionInfo ());
>     // Johan: Don't reset the local_id, it makes the structs for
>     // set nonunique
>     //local_id = 0;
>     // --
289a319,457
> // Johan: check the operator, make joinpoints on assignments
> void TrackerDog::pre_BinaryExpr (CT_BinaryExpr *node) {
> // Dont add if it's outside of the project
> if (!db ().Project ()->isBelow ((Unit*)node->token ()->belonging_to ())) {
> return;
> }
>
> // Extract operands and operator
> CTree *left = node->Son(0);
> Token *op = ((CT-Token*)node->Son(1))->token();
> CTree *right = node->Son(2);
>
> // Is this an assignment
> if (op->type() == Puma::TOK_ASSIGN) {

```

```

>
> CRecord *record = NULL;
> CT_SimpleName *var = NULL;
> // Is the left hand side of the assignment a simple name
> if (left->NodeName() == CT_SimpleName::NodeId()) {
>
>     // Gather information to use when creating the join point
>     var = (CT_SimpleName*)left;
>     // Get the class of which this variable is a member
>     record = var->Object()->AttributeInfo()->Record();
> }
> // Is the left hand side a member reference, or member pointer
> else if (left->NodeName() == CT_MembRefExpr::NodeId() ||
>         left->NodeName() == CT_MembPtrExpr::NodeId()) {
>
>     CT_BinaryExpr *expr = (CT_BinaryExpr*)left;
>     if (expr->Son(2)->NodeName() == CT_SimpleName::NodeId()) {
>         var = (CT_SimpleName*)expr->Son(2);
>         record = var->Object()->AttributeInfo()->Record();
>     }
> }
>
> // Only add this jpl if it is a member of a class,
> // the variable is set and
> // the expression was really inside a function (and not a default for instance)
> if (record && var && func_stack.top()) {
>     // Create the joinpoint
>     _world.new_set(var, op, right, node, func_stack.top(), local_id);
>     // Increase the local id, making the newly created join point location
>     // unique
>     local_id++;
>     // Visit the right hand side, there can be more joinpoints in it
>     visit(right);
>     // Don't visit the left hand side, no joinpoints should be available there
>     prune();
> }
> }
> }
>
> void TrackerDog::pre_SimpleName (CT_SimpleName *node) {
> if (check_variable(node)) {
>     // Do the actual adding
>     _world.new_get (node, node, func_stack.top(), local_id);
>     // Increase the id counter
>     local_id++;
> }
> }
>
> void TrackerDog::pre_MembRefPtrExpr (CT_BinaryExpr *node) {
> // Extract operands and operator
> CTree *left = node->Son(0);
> CTree *right = node->Son(2);
>
> if (right->NodeName() == CT_SimpleName::NodeId()) {
>     CT_SimpleName *variable = (CT_SimpleName*)right;
>     if (check_variable(variable)) {
>         _world.new_get (variable, node, func_stack.top(), local_id);
>         local_id++;
>         visit (left);
>         prune ();
>     }
> }
> }

```

```

> }
>
> bool TrackerDog::check_variable(CT_SimpleName *variable) {
> // Don't add if outside the project
> if (!db ().Project ()->isBelow ((Unit*)variable->token ()->belonging_to ())) {
> return false;
> }
> // Get object info
> CObjectInfo *obj_info = variable->Object ();
> // Check so it was present
> if (!obj_info) {
> return false;
> }
> CAttributeInfo *attrib_info = obj_info->AttributeInfo ();
> if (!attrib_info) {
> return false;
> }
> // Get the record of the class which this variable is a member of
> CRecord *record = attrib_info->Record ();
> // Only add variables that are members of a class
> if (!record) {
> return false;
> }
> // Check so that we are inside a function
> if (!func_stack.top()) {
> return false;
> }
> // Everythings ok
> return true;
> }
>
> void TrackerDog::pre_DerefExpr(CT_DerefExpr *deref) {
> }
>
> void TrackerDog::pre_AddrExpr(CT_AddrExpr *addr) {
> prune_Get(addr);
> }
>
> // Check if it's a field name right inside, if it is prune it
> // because taking the address or dereferencing a field is not a read.
> void TrackerDog::prune_Get(CT_UnaryExpr *unary) {
> CTree *expr = unary->Son(1);
> bool isSimpleName = expr->NodeName() == CT_SimpleName::NodeId();
> bool isMembRef = expr->NodeName() == CT_MembRefExpr::NodeId();
> bool isMembPtr = expr->NodeName() == CT_MembPtrExpr::NodeId();
>
> if (isSimpleName || isMembRef || isMembPtr) {
> prune();
> }
> }
> // --
>
diff -r from/AspectC++/TrackerDog.h to/AspectC++/TrackerDog.h
39d38
< CT_BinaryExpr *assignment;
63a63,72
> // Johan: Handle operators
> void pre_BinaryExpr (CT_BinaryExpr *node);
> void pre_SimpleName (CT_SimpleName *node);
> void pre_MembRefPtrExpr (CT_BinaryExpr *node);
> bool check_variable (CT_SimpleName *var);
> void pre_DerefExpr(CT_DerefExpr *deref);

```

```

> void pre_AddrExpr(CT_AddrExpr *addr);
> void prune_Get(CT_UnaryExpr *expr);
> // --
>
71c80
<   _tunit (tunit), _world (jpl1), assignment (0), last_init_declarator (0),
---
>   _tunit (tunit), _world (jpl1), last_init_declarator (0),
78c87,90
< void run () {
---
> void run () {
> //Johan: Init local id
> local_id = 0;
> // --
diff -r from/AspectC++/Transformer.cc to/AspectC++/Transformer.cc
628,630c630,634
<
< // update the project repository
< _repo.update (*advice_info, pc);
---
> // update the project repository
> // Johan: disabled
> /*_vm << "update repository" << endvm;
> _repo.update (*advice_info, pc);
> _vm << "repository updated" << endvm;*/
638c642,644
< ifct_defs << ends;
---
> ifct_defs << ends;
> _vm << "weave invocation functions" << endvm;
> //_vm << ifct_decls.str() << endvm << ifct_defs.str() << endvm;
640c646,647
<
< ifct_decls.str (), ifct_defs.str ());
---
> ifct_decls.str (), ifct_defs.str ());
> _vm << "invocation functions done" << endvm;
750c757,800
< _vm--;
---
> _vm--;
>
> //Johan: Weave set join points
> _vm << "Set Join Points" << endvm;
> _vm++;
> for (int i = 0; i < plan.set_jp_plans(); i++)
> {
> JPP_Code &jp_plan = plan.set_jp_plan (i);
> JPL_Set &jp_loc = plan.set_jp_loc (i);
>
> _vm << jp_loc.signature () << endvm;
>
> // create thisJoinPoint class
> jp_plan.jp_advice ()->mergeTJPFlags();
> if (jp_plan.jp_advice ()->tjp ().type_needed ()) {
> _code_weaver.make_tjp_struct(&jp_loc, jp_plan.jp_advice ());
> }
>
> // Weave the join point
> _code_weaver.set_join_point(&jp_loc, jp_plan);
> }
> _vm--;

```

```

>
> _vm << "Get Join Points" << endvm;
> _vm++;
> for (int i = 0; i < plan.get_jp_plans(); i++)
> {
>     JPP_Code &jp_plan = plan.get_jp_plan (i);
>     JPL_Get &jp_loc = plan.get_jp_loc (i);
>
>     _vm << jp_loc.signature () << endvm;
>
>     // create thisJoinPoint class
>     jp_plan.jp_advice ()->mergeTJPFlags();
>     if (jp_plan.jp_advice ()->tjp ().type_needed ()) {
>         _code_weaver.make_tjp_struct(&jp_loc, jp_plan.jp_advice ());
>     }
>
>     // Weave the join point
>     _code_weaver.get_join_point(&jp_loc, jp_plan);
> }
> _vm--;
>
> // --
diff -r from/AspectC++/WeaverBase.h to/AspectC++/WeaverBase.h
73c73
<
---
> public:

```

Bibliography

- [1] ACM digital library. Webpage.
<http://portal.acm.org/dl.cfm>.
- [2] Fault tolerant system. Wikipedia article.
http://en.wikipedia.org/wiki/Fault_tolerance.
- [3] AspectC++ mailing list. Webpage.
<http://www.aspectc.org/pipermail/aspectc-user/2005-May/000585.html>.
- [4] *AspectJ Documentation*.
<http://www.eclipse.org/aspectj/doc/released/progguide/semantics-joinPoints.html>.
- [5] AspectC++ Project Webpage. <http://www.aspectc.org/>.
- [6] Gnu general public license. Webpage, June 1991.
<http://www.gnu.org/copyleft/gpl.html>.
- [7] Survey of Alias Analysis. Webpage, October 2005.
<http://www.cs.princeton.edu/jqwu/Memory/survey.htm>.
- [8] Gnu diffutils. Webpage.
<http://www.gnu.org/software/diffutils/diffutils.html>.
- [9] Guidelines for the use of the c language in vehicle based software. The Motor Industry Software Reliability Association, April 1998.
- [10] AspectJ. Webpage.
<http://www.aspectj.org/>.
- [11] R. Alexandersson, P. Öhman, and M. Ivarsson. Aspect oriented software implemented node level fault tolerance. In *Ninth IASTED International Conference on Software Engineering and Applications (SEA 2005)*, Phoenix, AZ, USA, November 2005.
- [12] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and

- side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1993. ACM Press.
- [13] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.
- [14] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM Press.
- [15] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. AspectC++: Language Proposal and Prototype Implementation. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, October 2001.
- [16] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 24–34, New York, NY, USA, 2001. ACM Press.
- [17] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM Press.
- [18] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113–123, New York, NY, USA, 2000. ACM Press.
- [19] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.
- [20] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Company, 2003.
- [21] Gail C. Murphy, Robert J. Walker, Elisa L. A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten. Does aspect-oriented programming work? *Commun. ACM*, 44(10):75–77, 2001.
- [22] O. Spinczyk and M. Urban. The puma project webpage. <http://ivs.cs.uni-magdeburg.de/puma/>.
- [23] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on*

Principles of programming languages, pages 32–41, New York, NY, USA, 1996. ACM Press.

- [24] M. Urban and O. Spinczyk. *AspectC++ Language Reference*. pure-systems GmbH, 1.5 edition, May 2005. <http://www.aspectc.org/>.
- [25] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. An initial assessment of aspect-oriented programming. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 120–130, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
- [26] R. P. Wilson. *Efficient, context-sensitive pointer analysis for c programs*. PhD thesis, Stanford University, December 1997.