Compose*: a Language- and Platform-Independent Aspect Compiler for Composition Filters

A. de Roo, M. Hendriks, W. Havinga, P. Dürr, L. Bergmans
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract

This paper presents Compose*, a compilation and execution framework for the Composition Filters model. The Composition Filters model is designed to improve the composability of object-based programs. It is claimed that this approach is largely language-independent, and has previously been applied to languages such as Smalltalk, Java and C++. However, building a new Composition Filters compiler for each target language results in the duplication of compilation technology. Therefore, one of the aims of Compose* is to provide a language and platform independent compiler framework for the Composition Filters model. This paper provides the design rationale behind a tool architecture that supports this language and platform independence. It explains the use of language independent abstractions of the base program structure and how to include existing tools, such as compilers, to interface with the target language. The language and platform independence of Compose* has been verified by applying the compiler framework to multiple languages of the .NET platform, the Java language and platform and the C language.

Key words: Compiler implementation, Composition Filters, Aspect Oriented Programming, Software Composition

1. Introduction

A key design goal of the *Composition Filters model* is to improve the composability of programs written in object-based programming languages. The Composition Filters model has evolved from the first (published) version of the Sina language in the late 1980s [2,1], to a version that supports language independent composition of crosscutting concerns [22,5].

In this paper we discuss the Compose* tool, which implements the Composition Filters model for several languages and platforms. We will discuss the goals and design criteria for Compose*. We will very briefly discuss the key concepts of the Composition Filters model that are important for understanding this paper.

The Composition Filters model can be applied to object-based systems. In such a system, objects can send *messages* between each other, e.g. in the form of method calls or events. In the Composition Filters model, these messages can be filtered using a set of *filters*, as shown in figure 1. Each *filter* has a *filter type*, which defines the behavior that should be executed if the filter accepts the message and the behavior that should be executed if the filter rejects the message. The matching behavior of a filter is specified by a sequence of *filter expressions*, which offer a simple declarative language for state and message matching. Filters defining related functionality are grouped in so-called *filter modules*. Such filter modules can also encapsulate some internal state or share state with other objects.

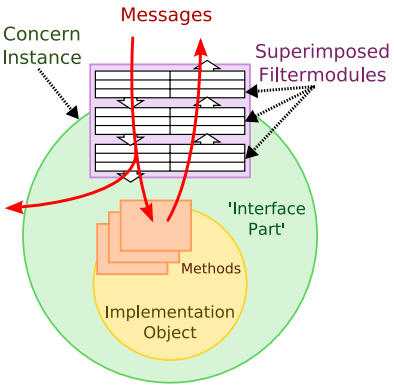

Fig. 1. Overview of the Composition Filters model

To indicate which filter modules should be applied (*superimposed*) to which objects, we use *superimposition selectors*. A superimposition selector selects a set of classes using a Prolog-based selector language. A specified filter module is applied to this selected set of classes. The result is that all messages sent to and received by all instances of those selected classes, are subjected to the filters within the filter module.

The Composition Filters model can be applied to many different languages, and historically we have done so, e.g. in SmallTalk [23], Java [24] and C++ [12]. Recently, we started to develop a new tool, called *Compose**, which not only supports .NET, but also Java and C. In this paper we discuss and report on the requirements and impact of these requirements on the development of this language and platform agnostic tooling.

Note that, although the Composition Filters model applies naturally to dynamically typed languages as exemplified by earlier implementations [21,23,3], this paper focuses on addressing issues that occur when creating tools for statically typed environments, such as .NET and Java.

This paper is organized as follows. Section 2 introduces the goals and design criteria of Compose*. Section 3 gives a high level overview of the architecture of Compose*. The design rationale of two criteria, language independence and platform independence, is explained in section 4. Section 5 discusses related work. Section 6 concludes the paper.

2. Goals and Criteria

The Compose* tool has been developed with two goals in mind:

(i) *To provide a framework to experiment with new language concepts & features*

(ii) *To provide the ability for researchers and practitioners to apply the Composition Filters language*

Besides these two goals for the tool, we list the four most important criteria for the design of Compose*:

(i) *Language independence*: We claim that the Composition Filters model can be applied to any programming language that supports the notion of message passing

between objects. Implementing a new compiler for each target language results, however, in the duplication of a substantial part of the Composition Filters specific implementation. Therefore, Compose* should implement this technology in a language independent way (as discussed before, we limit ourselves to statically typed languages). This results in a framework that can be used to create a language specific compiler for any target language.

(ii) *Platform independence*: Language independence is related to platform independence. Using Compose* for several platforms, like the .NET platform or the Java platform, implies that we cannot depend on the language tools, such as compilers, of one particular platform. Instead, Compose* should be able to operate with many different target language compilers and other language tools.

(iii) *Evolvability*: One goal of Compose* is to experiment with new concepts and features in the Composition Filters model. These new concepts and features are varying in nature, from new static analysis techniques to changes in the execution model of Composition Filters. The design of Compose* should facilitate experimentation by making it easy to extend the tool with new concepts and features.

(iv) *Performance*: Extending a certain programming language with the Composition Filters model will have an impact on the compile-time and runtime performance of the programming language. This performance overhead should be minimized. There are, however, trade-offs with the other criteria and, accordingly, trade-offs between the two goals for Compose*. For the first goal, evolvability is more important than performance. For the second goal, performance becomes of greater importance.

This paper focuses on the first two criteria.

3. Top-Level Process

The Compose* compiler is divided into two parts, a *platform independent* part (referred to as *Core*) and a *platform specific* part. Each part contains a set of modules that perform specific tasks of the compilation process. The top-level architectural design of the Compose* compiler is much like the design of modern compilers, where a shared repository is used by the various compiler parts [19]. A platform specific configuration determines what modules are used and in which order they are executed. Not all platforms use the same set of modules; this depends on the implementation for the specific platform.

Figure 2 provides a global overview of the compilation process and how it is divided in the Core and platform specific parts. The compiler input consists of the source files of the base program and concern files, which contain the composition filter specifications. The *type harvester* analyses the base program and produces a representation of the program in an abstract language model. It will also create stubs of the input, which are used in a later stage (section 4.1.3 explains stubs in detail). The input for the type harvester depends on the used platform. This could either be plain source files or assemblies, as long as a similar language model is produced. The *code generation* of the compiler has both a platform specific part and a general part. The general part of the code generation operates on the language and platform neutral composition filters. The platform specific part translates this to platform specific information. The language model and concern files are used to create a weave specification. The stubs are updated to reflect the updated interfaces of

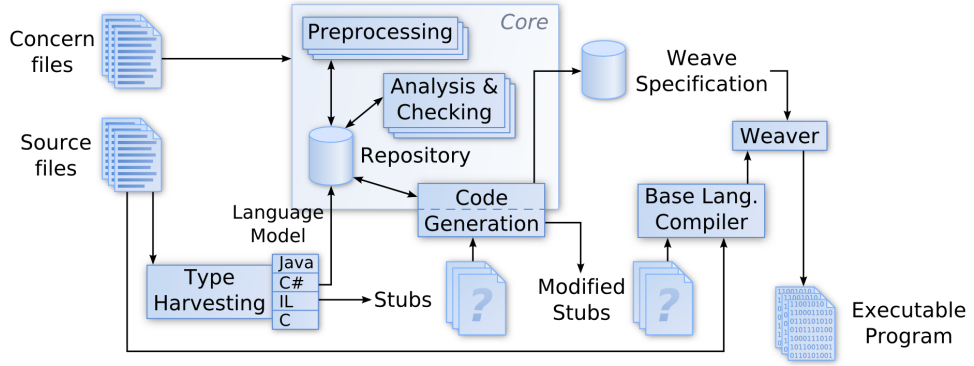

Fig. 2. Compose* compiler process

the types in the base program. The original source files are compiled against the modified stubs. The result of this process is used together with the weave specification to produce the final executable program. Included within the Core are a couple of modules that perform analysis and validation of the composition filter specifications. Some modules are not required to produce the executable program, so these could be disabled. The last part of the compilation process (concerning the Code Generation, Base Language Compiler, and Weaver) differs for the various platform implementations.

3.1. *Example Program*

To illustrate how Compose* works we use a simple program to which we apply the observer pattern. This example is used in section 4 to explain certain concepts. The program consists of a class `Shape` with the methods `setX` and `setY`. Using Compose* this class will be changed to become an observable class that sends a notification to its observers when the `setX` and `setY` are called. Listing 1 shows a concern definition that makes this change to the `Shape` class. Concern definitions do not contain any base language specific content.

A concern definition consist of two parts: filter module definitions and a superimposition block. A filter module contains the definitions when composition filters are applied to a message. The filter module in our example (lines 3 to 11) contains an *internal* declaration (line 6) and two filter definitions (lines 8 to 10). An internal creates a composition relation between the object on which this filter module is superimposed and an instance of the type declared in the internal. The internal can be used in the filter definitions as a destination for the message. The filter definition on lines 8 to 9 creates a *dispatch* filter that forwards a message to a new destination. In this case, messages matching `*.attach` and `*.detach` are forwarded to the internal `subject`. A message consists of a target and selector: `target.selector`. The target is the object that receives the message and the selector is the called method. The second filter in this filter module (line 10) defines an *after* filter. An after filter sends a new message after a given message has returned. In this case, the message `subject.notify` will be dispatched when a call to the methods `setX` or `setY` has returned. The superimposition block (lines 13 to 19) determines which filter modules will be superimposed to selected classes. The selector definition (line 16) selects

```
1  concern ObserverPattern
2  {
3    filtermodule Observable
4    {
5      internals
6        subject : Subject;
7      inputfilters
8        atdet : Dispatch = { [*.attach] subject.attach,
9                             [*.detach] subject.detach };
10       notif : After    = { {[*.setX], [*.setY]} subject.notify }
11   }
12
13   superimposition
14   {
15     selectors
16       shapes = { C | isClassWithName(C, 'Shape') };
17     filtermodules
18       shapes <- Observable;
19   }
20 }
```

Listing 1. Example concern specification

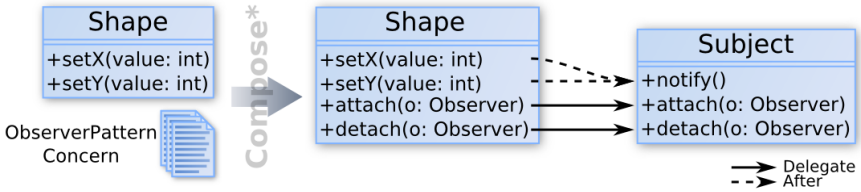

Fig. 3. Transformation of the example program

a collection of program elements; in this example it selects all classes with the name `Shape`. The filter module `Observable` is superimposed on all program elements selected by the selector `shapes` on line 18. The filter definitions of a filter module form an *advice*. The *pointcuts* are determined by the selectors and message matching in the filter definitions.

The concern definition in this example creates a composition between the class `Shape` and the class `Subject`. The `Subject` class implements the logic for an observable type, it manages the list of observers and sends notifications to them. Each `Shape` is associated with a `Subject` instance. The result of this composition is shown in figure 3.

The `Shape` class is extended to contain the methods `attach` and `detach`. The execution of these methods is delegated to the `Subject` class. This change makes the following statement possible: `myShape.attach(new MyObserver());`. Thus the interface of the class `shape` has been extended in a way similar to AspectJ's inter-type declarations. When the call to the methods `setX` and `setY` returns, a message is issued to `Subject.notify`, which will notify all observers about a change in the `Shape`.

The Core will perform the transformation on the language independent model of the base program which was created by the type harvester. The platform and language dependent parts of Compose* will effectuate the transformations into an executable program. The next section explains how these transformations are made possible. The above con-

cern specification can be used for any program written in any (supported) language that has a similar model for the `Shape`, `Subject` and `Observer` classes.

4. Design Rationale per Criterion

4.1. *Language Independence*

A major design goal for Compose* is to build tooling that is language-independent to the extent possible, viz., that shares functionality between different languages as much as possible.

In addition, to avoid the duplication of existing functionality, we prefer to reuse existing tools (such as parsers or compilers) for particular target languages. However, close integration (at the source code level) with existing tools typically leads to tight coupling, thus making it harder to write tooling that would be reusable in different languages. In addition, source code access to existing tooling is not always available.

Therefore, our approach is to use existing compilers for the target language, and then modify its outputs (e.g., class definitions in bytecode or similar intermediate language) according to the composition filter definitions.

Such an approach leaves several issues to be addressed:

(i) *Language-dependent notion of message passing:* at the behavioral level, the Composition Filters model assumes the notion of message passing between certain entities. Thus, we need to decide what comprises the "basic entities" and "message passing mechanism" for particular languages.

(ii) *Selecting where to superimpose aspect behavior ("pointcuts"):* the structure of the program can be used as a criterion to specify where (i.e., to which entities) particular composition filters should be applied. Thus, a structural model of programs expressed in the target language is required, so that it can be "queried" by a pointcut language.

(iii) *Object interface extension:* a major issue when designing aspect language compilers is the potential influence of aspect specifications on the type system of the target language. If aspect languages offer additional (static) composition mechanisms, aspect languages have to augment the existing type system. Especially in the case of statically typed languages, this may not be straightforward. As an example, consider intertype declarations in AspectJ, which influence the object interface of classes to which new methods are introduced. The Java type system has to be augmented to accommodate this; to be precise, calling a regular Java compiler on the Java part of the program will lead to type checking errors if base classes attempt to make calls to methods introduced by aspects. AspectJ augments the Java compiler, solving this problem by adding introduced methods to the program AST before the Java part of the compiler type checks the program. Compose* also supports mechanisms that effectively extend the interface of existing objects. As we want to reuse existing target language compilers without modification, this poses a problem when trying to compile the non-aspect part of (statically typed) programs.

We discuss our approach to solve these issues below.

4.1.1. *Language-Dependent Notion of Message Passing*

In the Composition Filters approach, if a filter module is applied to an entity, messages sent from or to that entity are passed through a set of filters as specified by the filter module.

Thus, in the Compose* tool implementation, language-specific notions of "messages" between "entities" have to be defined. In the case of object-oriented languages, such as Java or C#, *method calls* between *objects* map naturally to this model. In a procedural setting, such as the C language, we model *function calls* as messages, while groups of functions (for example, *.c-files* are often used to group related functionality in C programs) are considered the basic object entities. Thus, in the C version of Compose*, filter modules can be imposed on calls to (or from) functions in a particular group (e.g., *.c-file*).

In the Observer example presented in listing 1 and figure 3, two messages sent to objects of type `Shape` are `setX` and `setY`. The `After` filter on line 10 accepts these messages and thus adds behavior after the execution of the base methods. The `After` behavior is implemented by the method `notify` in `Subject`.

4.1.2. *Selecting where to Superimpose Aspect Behavior*

In the Composition Filters model, *superimposition selectors* ("pointcuts") are used to specify where aspect behavior should be applied. Thus, a model of the structure of the program (e.g., an Abstract Syntax Tree) is required, containing sufficient detail to support the elements that can be "queried" by the pointcut language. We have defined a *common structural language model* for class-based object-oriented languages, so that the same kind of queries can apply to programs written in languages that fit this model.

In Compose*, we use a general purpose logic-based language (prolog) to express pointcuts [13]. The program structure is presented to the prolog inference engine as a series of facts about each "program element" (class, method, etc.), as well as the relations between them. This uniform interface is defined as a view on the common structural language model mentioned above. Pointcuts are then essentially logic queries over this fact base. An example of such a logic query is shown on line 16 of listing 1. In this case, all classes with name `Shape` are selected (which effectively is one class). On line 18, the `Observable` filter module is superimposed on this set of classes.

To obtain such structural program models, we adopted two methods that do not rely on integration with an existing compiler for the target language. The first is to create a (partial) parser and AST-builder for the language, at least to the extent that is needed by the pointcut language. The second option is to compile the program using an existing compiler for the target language, then "harvesting" all necessary type information from the compiled (bytecode/intermediate language) version using either reflection or an Intermediate Language analyzer. If feasible, this method is preferred, because it does not require a parser for each language that maps to a particular intermediate language (IL). In the case of .NET, several languages map to the same IL "platform", so we can reuse the same *type harvester* for all of them.

4.1.3. *Dealing with Object Interface Extension*

The possibility to extend object interfaces is an important feature of the Composition Filters model, as it allows the implementation of alternative composition strategies not

supported by many OO languages, such as explicit delegation or multiple inheritance.

In the Composition Filters approach, a filter may "accept" a message selector (method name) that was not previously defined within the class(es) on which the filter is imposed. This means it is possible to send messages (method calls) that were not previously supported by the receiving class, thus effectively extending its interface. An example of this is shown in listing 1. The `Dispatch` filter on lines 8 and 9 dispatches the (previously unsupported) messages `attach` and `detach` to the existing methods `attach` and `detach` in `Subject`. So, the interface of `Shape` should be extended with the methods `attach` and `detach`.

Such additional composition mechanisms are not recognized by the target language compiler, and thus lead to compile errors when "introduced" methods are called. One could consider simply adding those method definitions to the base source code before compiling the base code; however, to do so, the aspects that decide which methods are introduced would have to be evaluated first. To evaluate the aspects, Compose* needs type information about the program. If the type harvester (as described in the previous subsection) depends on a compiled version of the program, this obviously does not work, as the base program cannot yet be compiled. Besides, the ways in which the base source code would have to be modified is likely to be highly language specific.

To deal with this, we take the following approach. First, the aspect compiler generates "*stub*" classes (classes with only empty method definitions) for every class in the program. Listing 2 shows the "stub" class for `Shape` from the Observer example. Fortunately, the information needed to generate these "stubs" can be obtained in a relatively language-independent way: IDE's such as Visual Studio (for .NET) or Eclipse (for Java) support interfaces through which such information can be extracted in a uniform way. Unfortunately, this creates a dependency on these IDE's; in addition, not all .NET languages actually support these interfaces completely or correctly. Therefore, we have in some cases implemented our own partial parsers to generate these "stub" classes, while at the same time generating the structural language model as described in the previous subsection.

Regardless of the exact approach used, the important point is that they do not require the source code to compile correctly – it is sufficient that it can be parsed.

```
1 public class Shape{
2   public void setX(int value){}
3   public void setY(int value){}
4 }
```

Listing 2. The "stub" version of class `Shape`

Once such "stub" classes are generated using one of the methods described above, they can be used to gather the type information that is necessary to evaluate the aspects. During the code generation phase, Compose* calculates the set of methods supported by each class - potentially adding new methods that can be called because a filter specification will handle them. The "stub" definitions are then updated to take these new methods into account, as shown in listing 3 for class `Shape`. Subsequently, the existing target language compiler can be used to compile each original (base) class against the "stub" versions of other classes that it depends on. By compiling against these "stub" versions, calls to methods that are not (yet) actually implemented now compile as well. Finally, the weaver modifies the (compiled) classes to take the aspect-defined behavior into account.

The weaving process is described in more detail in the following section.

```
public class Shape{
  public void setX(int value){}
  public void setY(int value){}
  public void attach(Observer o){}
  public void detach(Observer o){}
}
```

Listing 3. The modified "stub" version of class `Shape`

4.2. *Platform Independence*

There is a close relation between language independence and platform independence. Platform, as used within Compose*, relates to the execution environment of the base program, and the availability of tools or methods to apply composition filters to the base program.

The .NET programming languages all compile to the common intermediate language (CIL) of the .NET Framework. Operating on the CIL allows Compose* to modify any program written in a .NET language in a uniform way. The base program could also be modified at runtime using a modified virtual machine. But this would require to develop and maintain a custom implementation of the virtual machine, which is not within the scope of the Compose* project.

The requirement for language independence could easily be satisfied by only supporting .NET languages. But then one could argue that Compose* only supports a single (machine) language (i.e. CIL). Supporting a single platform could also lead to architectural designs that are closely bound to the tools and methods of that platform. Platform independence strengthens the language independence.

For each platform the following problems need to be resolved:

(i) *Execution of the composition filters within the program:* The composition filters add new logic to the base program, which changes its behavior. The composition filter specification cannot be directly executed by the target platform. So, this specification needs to be translated into an executable format (e.g. *code generation*).

(ii) *Changing the base program to integrate the composition filter advices:* The second part of the problem is to merge the executable format of the composition filters with the base program at the correct locations. This process is called *weaving*.

4.2.1. *Composition Filter Translation*

There are multiple ways to translate the composition filters to executable code, ranging from completely dynamic to completely static.

Dynamic execution allows the composition filters to be modified at runtime. To support dynamic execution, a *runtime interpreter* is needed to evaluate the composition filter specification. This runtime interpreter needs to be called from the locations where the composition filters are applied to. Depending on the amount of desired dynamic execution, changes to the virtual machine are required. As mentioned previously, changing a virtual machine is not within the scope of Compose*.

With static execution, the composition filter code is translated to the target language and merged with the base program. In this case, the composition filters becomes an integral part of the base program. Static execution of the composition filters offers less flexibility and evolvability than dynamic execution, but it does offer better runtime performance.

Compose* has two implementations of composition filter translation, a completely static implementation (*inliner*) and a partially dynamic implementation in the form of a runtime interpreter. In the implemented runtime interpreter only the message handling is interpreted. The superimposition is static.

4.2.2. *Weaving Process*

For the runtime interpreter approach, a platform specific weaver weaves interceptor calls to the interpreter at each "join point shadow". There is no platform independent part for this approach.

The inliner approach is divided into two stages, to optimize platform independence. In the first stage, the composition filters are translated to specific advice code for each join point shadow. The advice code is represented in a platform and language independent *procedural control flow model*. This model closely resembles the language paradigms of the target platforms, which makes it straightforward to translate it to platform specific code. Platform specific weavers translate the code model to the target language and weave the code in the base program.

Listing 4 shows in pseudo code [1] how the filters from the Observer example are translated to code and woven in the `Shape` class.

```
1  public class Shape{
2    ...
3    private Subject subject = new Subject();
4
5
6    public void setX(int value) {
7      // Start filter code
8      if (!<inner call flag set>){
9        // recursive call to execute base code:
10       <set inner call flag>
11       this.setX(value);
12
13       // After advice:
14       subject.notify();
15       return;
16     }
17     <reset inner call flag>
18     // End filter code
19
20     <base code>
21   }
22
23
24   public void setY(int value) {
```

[1] For readability, this pseudo code is written in a java-like language. Real weaving will generally be done on an intermediate language, like java bytecode.

```
25      // Start filter code
26      if (!<inner call flag set>){
27        // recursive call to execute base code:
28        <set inner call flag>
29        this.setY(value);
30
31        // After advice:
32        subject.notify();
33        return;
34      }
35      <reset inner call flag>
36      // End filter code
37
38      <base code>
39    }
40
41    public void attach(Observer o) {
42      subject.attach(o);
43    }
44
45    public void detach(Observer o) {
46      subject.detach(o);
47    }
48  }
```

Listing 4. Code generation and weaving in the `Shape` class

The *internal* `subject` is woven as an instance variable `subject` (Line 3).

The `After` filter that matches `setX` and `setY` is translated to a block of filter code. The `After` advice needs to be executed after the execution of the base code. Because the base code might have multiple exit points, weaving the `After` advice is not trivial. We implemented this in the following way: The block of filter code is inserted in the method in front of the base code. To select between the execution of the filter code and the base code, an inner call flag is used (the implementation details are not shown). This inner call flag is normally not set, which leads to the execution of the filter code (see the check on line 8). The `After` advice should be executed after the execution of the base code. Therefore, first a recursive call is made with the inner call flag set, to execute the base code (lines 10-11). When the recursive call to the base code has returned, the `After` advice will be executed by calling `subject.notify` (line 14). The execution of the filter code ends with a return (line 15).

We choose this type of weaving and not weaving on each exit point, for example, to prevent code duplication and to have a consistent weaving strategy for all filter types (the behavior of all filter types can be located into a single filter code block). More details about this solution can be found in [8].

The `Dispatch` filter that matches `attach` and `detach` is translated to a delegation method call to `subject.attach` respectively `subject.detach`.

5. Related Work

The GNU Compiler Collection (GCC) [11] is a compiler framework that supports multiple programming languages and target platforms. GCC is set up such that it is easy

to add support for new programming languages, which can be compiled to any of the supported platforms. It is also possible to add new platforms, to which all supported programming languages can be compiled. To accomplish this GCC makes use of a generic intermediate language (GIMPLE) to which all supported languages are translated. Compose* resembles GCC in the sense that both have a platform and language independent core that performs most of the processing, in addition to language-specific front ends (for parsing/type harvesting) and platform specific back-ends (for weaving/code generation). However, in Compose* the language and platform support are not independent from each other as is the case within GCC.

Reflex [20] is an AOP kernel providing a framework that facilitates the implementation of multiple AOP languages on top of Java. In contrast, Compose* is an implementation of a specific approach to AOP (i.e., the Composition Filters model), and applies this model to several target languages and platforms.

The Aspectbench compiler, or "abc" [4] is an alternative compiler for the AspectJ language. Like Compose*, it is set up as a compiler framework that allows for extensions, thus aiming to facilitate the implementation of experimental language features. The project focuses on extending the AspectJ language, as well as providing optimized implementations of the language. As compared to Compose*, abc does not try to address multiple target languages or platforms.

Weave.NET [16,15] describes a weaver for .NET that supports multiple target languages. Aspects are specified using XML notation, the specifications are based on a subset of the joinpoint types and matching patterns used in AspectJ. As compared to this, the Composition Filters model uses a more expressive language, and in addition Compose* implements several static analysis and checking modules to check aspects for potential interference (with each other or the base program). Also, Compose* targets several platforms and supports several weaving strategies, such as code inlining, or based on runtime interpretation of filter specifications.

XWeaver [6,18] is an aspect weaver for C/C++ and Java where weaving is performed at the source code level. XWeaver uses srcML [7] to convert the input source into an XML representation. On this XML representation transformations are performed according to the instructions as defined in XWeaver's aspect language: AspectX. The code modifications defined in the AspectX sources are base language specific; they often contain code fragments written in the base language. Because of this, XWeaver/AspectX cannot truly be considered a language independent aspect weaver and aspect language.

6. Discussion and Conclusion

Compose* is a language and platform independent compiler framework for the aspect oriented Composition Filters model. The Core of the framework contains all language independent Composition Filter compilation technology, like superimposition resolving and static analysis techniques. We have shown how the Core is made language independent (of the target language) by using a generic language model that contains the representation of the base program structure.

One of the design goals of the system is to try to put as much functionality as possible in the –generic– Core, to avoid reimplementation of the same functionality for multiple languages or platforms. Most of the innovations of the Compose* language are imple-

mented in the Core. For example, the filter module (advice) ordering specifications [2] [17], the algorithm for detecting ambiguous inter-type declarations [14] and (run-time) aspect interference detection [9,10]. The architecture of the compiler has enabled the incremental implementation of such innovative features in a highly modular way.

To apply the framework to a specific target language, a type harvester and a weaver need to be built. The type harvester is used to extract the language model information from the base program. The weaver then modifies the base program to apply the composition filters. To create these components, existing tools are used as much as possible, instead of implementing dedicated solutions. Adopting existing tools may cause difficulties, for example type checking problems. We have shown how to cope with these difficulties.

We have developed Compose* implementations for the .NET platform, the Java platform and the C language. Table 1 shows for each implementation the total size of the implementation, the size of the language and platform specific part of the implementation and the relative size of the reused Core in the implementation. The table shows that the reused Core occupies a relatively large part in each implementation, e.g. relatively little effort is needed to build an implementation of Compose* for a specific platform. This confirms the language and platform independence of Compose*.

Platform	Total SLOC	Platform SLOC [3]	% Core
Core [4]	39908	-	-
Runtime Interpreter Core [5]	6838	-	-
.NET (interpreter)	56604	12190	78%
.NET (inliner)	58543	18635	68%
Java (interpreter)	49208	4794	90%
C (inliner)	43930	4022	91%

Table 1: Physical source lines of code of Compose* platforms

References

[1] M. Akşit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In O. L. Madsen, editor, *Proc. 7th European Conf. Object-Oriented Programming*, pages 372–395. Springer-Verlag Lecture Notes in Computer Science, 1992.

[2] Although part of these is performed at run-time, hence affecting the interpreter/inliner

[3] This includes only the code to create the compiler for the specific platform. Code for other functionality, such as IDE integration, has been excluded.

[4] This contains all Core compiler code, including the platform independent inliner code. The latter is not included with the interpreted platforms.

[5] The runtime interpreter is written in Java and is source compatible with VisualJ#.

[2] M. Akşit and A. Tripathi. Data abstraction mechanisms in sina/st. In *Proceedings of the conference Object-Oriented Systems, Languages and Applications*, volume 23 of *ACM Sigplan Notices*, pages 267–275, 1988.

[3] M. Aksit. *On the Design of the Object-Oriented Language Sina*. PhD thesis, University of Twente, Mar 1989.

[4] P. Avgustinov, A. Christensen, L. Hendren, and S. Kuzins. abc: an extensible aspectj compiler. In *Proceedings of the 4th conference on Aspect Oriented Software Development, AOSD 2005*, Jan 2005.

[5] L. Bergmans and M. Akşit. Principles and design rationale of composition filters. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 63–95. Addison-Wesley, Boston, 2005.

[6] I. Birrer, P. Chevalley, A. Pasetti, and O. Rohlik. An Aspect Weaver For Qualifiable Applications. *Proceedings of the 14-thDigital Avionics Systems in Aerospace (DASIA)*, 2004.

[7] M. Collard. Addressing source code using srcml. *IEEE International Workshop on Program Comprehension Working Session: Textual Views of Source Code to Support Comprehension (IWPC05)*, 2005.

[8] A. J. de Roo. Towards more robust advice: Message flow analysis for composition filters and its application. Master's thesis, University of Twente, The Netherlands, Mar. 2007.

[9] P. Dürr. *Resource-based Verification for Robust Composition of Aspects*. PhD thesis, University of Twente, June 2008.

[10] P. E. A. Durr, L. M. J. Bergmans, and M. A. sit. Static and dynamic detection of behavioral conflicts between aspects. In O. Sokolsky and S. Tasiran, editors, *Proceedings of the Seventh International Workshop on Runtime Verification, Vancouver, Canada*, volume 4839 of *Lecture Notes in Computer Science*, pages 38–50, Berlin, December 2007. Springer Verlag.

[11] GCC team. GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[12] M. Glandrup. Extending C++ using the concepts of composition filters. Master's thesis, University of Twente, 1995.

[13] W. K. Havinga. Designating join points in composestar - a predicate-based superimposition selector language for compose*. Master's thesis, University of Twente, The Netherlands, May 2005.

[14] W. K. Havinga, I. Nagy, L. M. J. Bergmans, and M. A. sit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development, Bonn, Germany*, pages 214–225, New York, 2006. ACM Press.

[15] D. Lafferty. *Aspect-based Properties*. PhD thesis, University of Dublin, Trinity College, Oct 2004.

[16] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *Proceedings of the OOPSLA 2003 conference*, Jan 2003.

[17] I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, editors, *Proceedings of International Conference NetObjectDays, NODe2005*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, Sep 2005. Gesellschaft für Informatik (GI).

[18] P&P Software. XWeaver. http://www.xweaver.org.

[19] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1996.

[20] E. Tanter and J. Noyé. A versatile kernel for multi-language aop. In *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GPCE 2005*, Jan 2005.

[21] A. Tripathi, E. Berge, and M. Akşit. An implementation of the object-oriented concurrent programming language sina. In *Software Practice and Experience 19*, volume 3, pages 235–256, 1989.

[22] University of Twente. COMPOSE*. http://composestar.sourceforge.net.

[23] W. van Dijk and J. Mordhorst. CFIST, Composition Filters in Smalltalk. Graduation Report, HIO Enschede, The Netherlands, May 1995.

[24] J. C. Wichman. The development of a preprocessor to facilitate composition filters in the Java language. Master's thesis, University of Twente, 1999.