Incremental Compilation in Compose⋆

A thesis submitted for the degree
of Master of Science at
the University of Twente

Dennis Spenkelink

Enschede, October 28, 2006

Research Group

Twente Research and Education
on Software Engineering
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente

Abstract

Compose⋆ is a project that provides aspect-oriented programming for object-oriented languages by using the composition filters model. In particular, Compose⋆.NET is an implementation of Compose⋆ that provides aspect-oriented programming to Microsoft Visual Studio languages.

The compilation process of Compose⋆.NET contains multiple compilation modules. Each of them has their own responsibilities and duties such as parsing, analysis tasks and weaving. However, all of them suffer from the same problem. They do not support any kind of incrementality. This means that they perform all of their operations in each compilation without taking advantage of the results and efforts of previous compilations. This unavoidably results in compilations containing many redundant repeats of operations, which slow down compilation. To minimize these redundant operations and hence, speed up the compilation, Compose⋆ needs an incremental compilation process.

Therefore, we have developed a new compilation module called INCRE. This compilation module provides incremental performance as a service to all other compilation modules of Compose⋆. This thesis describes in detail the design and implementation of this new compilation module and evaluates its performance by charts of tests.

Acknowledgements

My graduation time was a long but exciting process and I would not have missed it for the world. Many people contributed to the completion of this thesis, for which I am grateful. In particular, I would like to express my appreciation to the following people.

First, I would like to thank my supervisor Lodewijk Bergmans for his enthusiastic and expert guiding. In addition, I am thankful to Gurcan Gulesir and István Nagy for their assistance in writing this thesis. Furthermore, I thank my fellow members of the Compose⋆ project for their valuable comments and tips. I learned a lot from you guys. Finally, many thanks go to my family for encouraging me to do my best and supporting me all the way.

Contents

List of Figures

Listings

Reading Guide

This page presents a brief guide to reading this thesis.

Chapter 1 provides a general introduction of Aspect-Oriented Software Development (AOSD). We recommend you to read this chapter if you are unfamiliar with concepts such as AOP, AspectJ and Composition Filters.

Chapter 2 gives insight into our implementation of the composition filters approach called Compose⋆. If you are already familiar with the internals of Compose⋆, you can skip this chapter and proceed to the next chapter. If not, then we advice you to read this chapter because it provides the necessary background information to understand the remaining chapters.

Chapter 3 identifies the goal of this thesis, which is providing incremental compilation to Compose⋆. Chapter 4 presents two solution approaches to achieve this goal. Further, it compares the two solution approaches and selects the most desirable one. Based on the chosen solution approach, chapter 5 compares several design alternatives and selects the most desirable design.

The compilation process of Compose⋆ contains multiple compilation modules. Each of them has their own responsibilities and duties. To bring incremental compilation to Compose⋆, we have developed a new compilation module called INCRE. This compilation module provides incremental performance as a service to all other compilation modules. Chapter 6 describes our implementation efforts of INCRE. Therefore, it uses Unified Modeling Language (UML) diagrams. If you are not familiar with UML concepts, we advice you to visit UML's official resource page[1].

In chapter 7, we describe in detail how eight existing non-incremental compilation modules have been adapted to use the incremental performance service provided by INCRE.

Chapter 8 evaluates the efficiency of the developed incremental compiler by means of charts. Based on these charts, we point out the most important benefits and limitations of the incremental compiler. Finally, chapter 9 summarizes the main findings of this thesis and identifies future and related work.

You can find more information on the Compose⋆ project and the incremental compiler at the website of Compose⋆ [2]. You can browse the source code of the incremental compiler online at SourceForge[3].

[1] http://www.uml.org/

[2] http://composestar.sf.net

[3] http://svn.sourceforge.net/viewvc/composestar/home/dspenkel/code

Chapter 1

Introduction to AOSD

The first two chapters have originally been written by seven M. Sc. students [27, 16, 59, 11, 52, 25, 10] at the University of Twente. The chapters have been rewritten for use in the following theses: [58, 12, 55, 30, 15, 28] and this thesis. They serve as a general introduction into Aspect-Oriented Software Development and Compose⋆ in particular.

1.1 Introduction

The goal of software engineering is to solve a problem by implementing a software system. The things of interest are called concerns. They exist at every level of the engineering process. A recurrent theme in engineering is that of modularization: separation and localization of concerns. The goal of modularization is to create maintainable and reusable software. A programming language is used to implement concerns.

Fifteen years ago, the dominant programming language paradigm was procedural program-

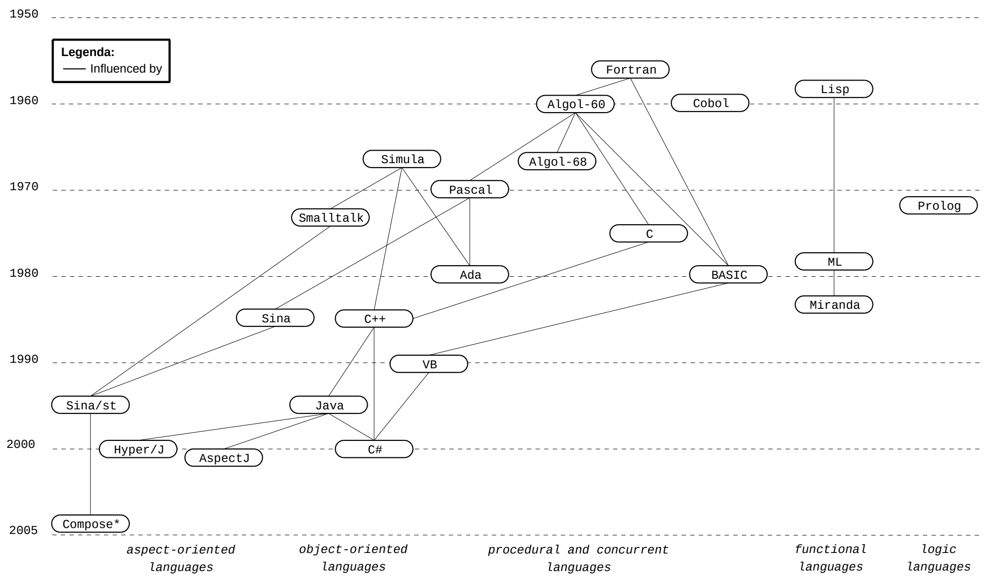


Figure 1.1: Dates and ancestry of several important languages

ming. This paradigm is characterized by the use of statements that update state variables. Examples are Algol-like languages such as Pascal, C, and Fortran.

Other programming paradigms are the functional, logic, object-oriented, and aspect-oriented paradigms. Figure 1.1 summarizes the dates and ancestry of several important languages [60]. Every paradigm uses a different modularization mechanism for separating concerns into modules.

Functional languages try to solve problems without resorting to variables. These languages are entirely based on functions over lists and trees. Lisp and Miranda are examples of functional languages.

A logic language is based on a subset of mathematical logic. The computer is programmed to infer relationships between values, rather than to compute output values from input values. Prolog is currently the most used logic language [60].

A shortcoming of procedural programming is that global variables can potentially be accessed and updated by any part of the program. This can result in unmanageable programs because no module that accesses a global variable can be understood independently from other modules that also access that global variable.

The Object-Oriented Programming (OOP) paradigm improves modularity by encapsulating data with methods inside objects. The data may only be accessed indirectly, by calling the associated methods. Although the concept appeared in the seventies, it took twenty years to become popular [60]. The most well known object-oriented languages are C++, Java, C#, and Smalltalk.

The hard part about object-oriented design is decomposing a system into objects. The task is difficult because many factors come into play: encapsulation, granularity, dependency, adaptability, reusability, and others. They all influence the decomposition, often in conflicting ways [21].

Existing modularization mechanisms typically support only a small set of decompositions and usually only a single dominant modularization at a time. This is known as the tyranny of the dominant decomposition [54]. A specific decomposition limits the ability to implement other concerns in a modular way. For example, OOP modularizes concerns in classes and only fixed relations are possible. Implementing a concern in a class might prevent another concern from being implemented as a class.

Aspect-Oriented Programming (AOP) is a paradigm that solves this problem.

AOP is commonly used in combination with OOP but can be applied to other paradigms as well. The following sections introduce an example to demonstrate the problems that may arise with OOP and show how AOP can solve this. Finally, we look at three particular AOP methodologies in more detail.

1.2 Traditional Approach

Consider an application containing an object `Add` and an object `CalcDisplay`. `Add` inherits from the abstract class `Calculation` and implements its method `execute(a, b)`. It performs the addition of two integers. `CalcDisplay` receives an update from `Add` if a calculation is finished and prints the result to screen. Suppose all method calls need to be traced. The objects use a `Tracer` object to write messages about the program execution to screen. This is implemented

```
1  public class Add extends Calculation{
2
3    private int result;
4    private CalcDisplay calcDisplay;
5    private Tracer trace;
6
7    Add() {
8      result = 0;
9      calcDisplay = new CalcDisplay();
10     trace = new Tracer();
11   }
12
13   public void execute(int a, int b) {
14     trace.write("void Add.execute(int, int
          )");
15     result = a + b;
16     calcDisplay.update(result);
17   }
18
19   public int getLastResult() {
20     trace.write("int Add.getLastResult()")
          ;
21     return result;
22   }
23 }
```

(a) Addition

```
1  public class CalcDisplay {
2    private Tracer trace;
3
4    public CalcDisplay() {
5      trace = new Tracer();
6    }
7
8    public void update(int value){
9      trace.write("void CalcDisplay.update(
          int)");
10     System.out.println("Printing new value
          of calculation: "+value);
11   }
12 }
```

(b) CalcDisplay

Listing 1.1: Modeling addition, display, and logging without using aspects

by a method called `write`. Three concerns can be recognized: addition, display, and tracing. The implementation might look something like Listing 1.1.

From our example, we recognize two forms of crosscutting: *code tangling* and *code scattering*.

The addition and display concerns are implemented in classes `Add` and `CalcDisplay` respectively. Tracing is implemented in the class `Tracer`, but also contains code in the other two classes (lines 5, 10, 14, and 20 in (a) and 2, 5, and 9 in (b)). If a concern is implemented across several classes, it is said to be scattered. In the example of Listing 1.1, the tracing concern is scattered.

Usually a scattered concern involves code *replication*. That is, the same code is implemented a number of times. In our example, the classes `Add` and `CalcDisplay` contain similar tracing code.

In class `Add` the code for the addition and tracing concerns are intermixed. In class `CalcDisplay` the code for the display and tracing concerns are intermixed. If more then one concern is implemented in a single class they are said to be tangled. In our example, the addition and tracing concerns are tangled. Also display and tracing concerns are tangled. Crosscutting code has the following consequences:

Code is difficult to change
: Changing a scattered concern requires us to modify the code in several places. Making modifications to a tangled concern class requires checking for side effects with all existing crosscutting concerns;

```
public class Add extends Calculation{
  private int result;
  private CalcDisplay calcDisplay;

  Add() {
    result = 0;
    calcDisplay = new CalcDisplay();
  }

  public void execute(int a, int b) {
    result = a + b;
    calcDisplay.update(result);
  }

  public int getLastResult() {
    return result;
  }
}
```

(a) Addition concern

```
aspect Tracing {
  Tracer trace = new Tracer();

  pointcut tracedCalls():
    call(* (Calculation+).*(..)) ||
    call(* CalcDisplay.*(..));

  before(): tracedCalls() {
    trace.write(thisJoinPoint.getSignature()
        .toString());
  }
}
```

(b) Tracing concern

Listing 1.2: Modeling addition, display, and logging with aspects

Code is harder to reuse
: To reuse an object in another system, it is necessary to either remove the tracing code or reuse the (same) tracer object in the new system;

Code is harder to understand
: Tangled code makes it difficult to see which code belongs to which concern.

1.3 AOP Approach

To solve the problems with crosscutting, several techniques are being researched that attempt to increase the expressiveness of the OO paradigm. Aspect-Oriented Programming (AOP) introduces a modular structure, the aspect, to capture the location and behavior of crosscutting concerns. Examples of Aspect-Oriented languages are Sina, AspectJ, Hyper/J, and Compose⋆. A special syntax is used to specify aspects and the way in which they are combined with regular objects. The fundamental goals of AOP are twofold [23]: first to provide a mechanism to express concerns that crosscut other components. Second to use this description to allow for the separation of concerns.

Join points are well-defined places in the structure or execution flow of a program where additional behavior can be attached. The most common join points are method calls. *Pointcuts* describe a set of join points. This allows us to execute behavior at many places in a program by one expression. *Advice* is the behavior executed at a join point.

In the example of Listing 1.2, the class `Add` does not contain any tracing code and only implements the addition concern. Class `CalcDisplay` also does not contain tracing code. In our example, the tracing aspect contains all the tracing code. The pointcut `tracedCalls` specifies at which locations tracing code is executed.

The crosscutting concern is explicitly captured in aspects instead of being embedded within the code of other objects. This has several advantages over the previous code.

Aspect code can be changed
Changing aspect code does not influence other concerns;

Aspect code can be reused
The coupling of aspects is done by defining pointcuts. In theory, this low coupling allows for reuse. In practice, reuse is still difficult;

Aspect code is easier to understand
A concern can be understood independent of other concerns;

Aspect pluggability
Enabling or disabling concerns becomes possible.

1.3.1 AOP Composition

AOP composition can be either symmetric or asymmetric. In the symmetric approach, every component can be composed with any other component. For instance, Hyper/J follows this approach.

In the asymmetric approach, the base program and aspects are distinguished. The base program is composed with the aspects. For instance, AspectJ (covered in more detail in the next section) follows this approach.

1.3.2 Aspect Weaving

The integration of components and aspects is called *aspect weaving*. There are three approaches to aspect weaving. The first and second approach rely on adding behavior in the program, either by weaving the aspect in the source code, or by weaving directly in the target language. The target language can be intermediate language (IL) or machine code. Examples of IL are Java byte code and Common Intermediate Language (CIL). The remainder of this chapter considers only intermediate language targets. The third approach relies on adapting the virtual machine. Each method is explained briefly in the following sections.

1.3.2.1 Source Code Weaving

The source code weaver combines the original source with aspect code. It interprets the defined aspects and combines them with the original source, generating input for the native compiler. For the native compiler there is no difference between source code with and without aspects. Hereafter the compiler generates an intermediate or machine language output (depending on the compiler-type).

The advantages of using source code weaving are:

High-level source modification
Since all modifications are done at source code level, there is no need to know the target (output) language of the native compiler;

Aspect and original source optimization
First, the aspects are woven into the source code and hereafter compiled by the native compiler. The produced target language has all the benefits of the native compiler opti-

mization passes. However, optimizations specific to exploiting aspect knowledge are not possible;

Native compiler portability
The native compiler can be replaced by any other compiler as long as it has the same input language. Replacing the compiler with a newer version or another target language can be done with little or no modification to the aspect weaver.

However, the drawbacks of source code weaving are:

Language dependency
Source code weaving is written explicitly for the syntax of the input language;

Limited expressiveness
Aspects are limited to the expressive power of the source language. For example, when using source code weaving, it is not possible to add multiple inheritance to a single inheritance language.

1.3.2.2 Intermediate Language Weaving

Weaving aspects through an intermediate language gives more control over the executable program and solves some issues as identified in subsubsection 1.3.2.1 on source code weaving. Weaving at this level allows for creating combinations of intermediate language constructs that cannot be expressed at the source code level. Although IL can be hard to understand, IL weaving has several advantages over source code weaving:

Programming language independence
All compilers generating the target IL output can be used;

More expressiveness
It is possible to create IL constructs that are not possible in the original programming language;

Source code independence
Can add aspects to programs and libraries without using the source code (which may not be available);

Adding aspects at load- or runtime
A special class loader or runtime environment can decide and do dynamic weaving. The aspect weaver adds a runtime environment into the program. How and when aspects can be added to the program depend on the implementation of the runtime environment.

However, IL weaving also has drawbacks that do not exist for source code weaving:

Hard to understand
Specific knowledge about the IL is needed;

More error-prone
Compiler optimization may cause unexpected results. Compiler can remove code that breaks the attached aspect (e.g., inlining of methods).

1.3.2.3 Adapting the Virtual Machine

Adapting the virtual machine (VM) removes the need to weave aspects. This technique has the same advantages as intermediate language weaving and can also overcome some of its disadvantages as mentioned in subsubsection 1.3.2.2. Aspects can be added without recompilation, redeployment, and restart of the application [49, 48].

Modifying the virtual machine also has its disadvantages:

Dependency on adapted virtual machines
: Using an adapted virtual machine requires that every system should be upgraded to that version;

Virtual machine optimization
: People have spent a lot of time optimizing virtual machines. By modifying the virtual machine these optimizations should be revisited. Reintegrating changes introduced by newer versions of the original virtual machine, might have substantial impact.

1.4 AOP Solutions

As the concept of AOP has been embraced as a useful extension to classic programming, different AOP solutions have been developed. Each solution has one or more implementations to demonstrate how the solution is to be used. As described by [18] these differ primarily in:

How aspects are specified
: Each technique uses its own aspect language to describe the concerns;

Composition mechanism
: Each technique provides its own composition mechanisms;

Implementation mechanism
: Whether components are determined statically at compile time or dynamically at run time, the support for verification of compositions, and the type of weaving.

Use of decoupling
: Should the writer of the main code be aware that aspects are applied to his code;

Supported software processes
: The overall process, techniques for reusability, analyzing aspect performance of aspects, is it possible to monitor performance, and is it possible to debug the aspects.

In the next sections, we introduce AspectJ [42], Hyperspaces [47] and Composition Filters [9], which are three main AOP approaches.

1.4.1 AspectJ Approach

AspectJ [42] is an aspect-oriented extension to the Java programming language. It is probably the most popular approach to AOP at the moment, and it is finding its way into the industrial software development. AspectJ has been developed by Gregor Kiczales at Xerox's PARC (Palo Alto Research Center). To encourage the growth of the AspectJ technology and community, PARC transferred AspectJ to an open Eclipse project. The popularity of AspectJ comes partly from the various extensions based on it. Various projects are porting AspectJ to other languages resulting in tools such as AspectR and AspectC.

One of the main goals in the design of AspectJ is to make it a compatible extension to Java. AspectJ tries to be compatible in four ways:

```
aspect DynamicCrosscuttingExample {
  Log log = new Log();

  pointcut traceMethods():
    execution(edu.utwente.trese.*.*(..));

  before() : traceMethods {
    log.write("Entering " + thisJointPoint.getSignature());
  }

  after() : traceMethods {
    log.write("Exiting " + thisJointPoint.getSignature());
  }
}
```

Listing 1.3: Example of dynamic crosscutting in AspectJ

Upward compatibility
All legal Java programs must be legal AspectJ programs;

Platform compatibility
All legal AspectJ programs must run on standard Java virtual machines;

Tool compatibility
It must be possible to extend existing tools to support AspectJ in a natural way; this includes IDEs, documentation tools and design tools;

Programmer compatibility
Programming with AspectJ must feel like a natural extension of programming with Java.

AspectJ extends Java with support for two kinds of crosscutting functionality. The first allows defining additional behavior to run at certain well-defined points in the execution of the program and is called the *dynamic crosscutting mechanism*. The other is called the *static crosscutting mechanism* and allows modifying the static structure of classes (methods and relationships between classes). The units of crosscutting implementation are called aspects. An example of an aspect specified in AspectJ is shown in Listing 1.3.

The points in the execution of a program where the crosscutting behavior is inserted are called *join points*. A *pointcut* has a set of join points. In Listing 1.3 is `traceMethods` an example of a pointcut definition. The pointcut includes all executions of any method that is in a class contained by package `edu.utwente.trese`.

The code that should execute at a given join point is declared in an advice. Advice is a method-like code body associated with a certain pointcut. AspectJ supports *before*, *after* and *around* advice that specifies where the additional code is to be inserted. In the example, both before and after advice are declared to run at the join points specified by the `traceMethods` pointcut.

Aspects can contain anything permitted in class declarations including definitions of pointcuts, advice and static crosscutting. For example, static crosscutting allows a programmer to add fields and methods to certain classes as shown in Listing 1.4.

The shown construct is called inter-type member declaration and adds a method `trace` to class `Log`. Other forms of inter-type declarations allow developers to declare the parents of classes (superclasses and realized interfaces), declare where exceptions need to be thrown, and allow a developer to define the precedence among aspects.

```
aspect StaticCrosscuttingExample {
  private int Log.trace(String traceMsg) {
    Log.write(" --- MARK --- " + traceMsg);
  }
}
```

Listing 1.4: Example of static crosscutting in AspectJ

With its variety of possibilities, AspectJ can be considered a useful approach for realizing software requirements.

1.4.2 Hyperspaces Approach

The *Hyperspaces* approach is developed by H. Ossher and P. Tarr at the IBM T.J. Watson Research Center. The Hyperspaces approach adopts the principle of multi-dimensional separation of concerns [47], which involves:

- Multiple, arbitrary dimensions of concerns;
- Simultaneous separation along these dimensions;
- Ability to dynamically handle new concerns and new dimensions of concern as they arise throughout the software life cycle;
- Overlapping and interacting concerns. It is appealing to think of many concerns as independent or orthogonal, but they rarely are in practice.

We explain the Hyperspaces approach by an example written in the *Hyper/J* language. Hyper/J is an implementation of the Hyperspaces approach for Java. It provides the ability to identify concerns, specify modules in terms of those concerns, and synthesize systems and components by integrating those modules. Hyper/J uses bytecode weaving on binary Java class files and generates new class files to be used for execution. Although the Hyper/J project seems abandoned and there has not been any update in the code or documentation for a while, we still mention it because the Hyperspaces approach offers a unique AOP solution.

As a first step, developers create hyperspaces by specifying a set of Java class files that contain the code units that populate the hyperspace. To do this is, you create a hyperspace specification, as demonstrated in Listing 1.5.

Hyper/J will automatically create a hyperspace with one dimension—the class file dimension. A dimension of concern is a set of concerns that are disjoint. The initial hyperspace will contain all units within the specified package. To create a new dimension you can specify concern mappings, which describe how existing units in the hyperspace relate to concerns in that dimension, as demonstrated in Listing 1.6.

The first line indicates that, by default, all of the units contained within the package `edu.utwente.trese.pacman` address the kernel concern of the feature dimension. The other map-

```
Hyperspace Pacman
  class edu.utwente.trese.pacman.*;
```

Listing 1.5: Creation of a hyperspace

```
package edu.utwente.trese.pacman: Feature.Kernel
operation trace: Feature.Logging
operation debug: Feature.Debugging
```

Listing 1.6: Specification of concern mappings

pings specify that any method named `trace` or `debug` address the logging and debugging concern respectively. These later mappings override the first one.

Hypermodules are based on concerns and consist of two parts. The first part specifies a set of hyperslices in terms of the concerns identified in the concern matrix. The second part specifies the integration relationships between the hyperslices. A hyperspace can contain several hypermodules realizing different modularizations of the same units. Systems can be composed in many ways from these hypermodules.

Listing 1.7 shows a hypermodule with two concerns, kernel and logging. They are related by a `mergeByName` integration relationship. This means that units in the different concerns correspond if they have the same name (`ByName`) and that these corresponding units are to be combined (`merge`). For example, all members of the corresponding classes are brought together into the composed class. The hypermodule results in a hyperslice that contains all the classes without the debugging feature; thus, no `debug` methods will be present.

The most important feature of the hyperspaces approach is the support for on-demand remodularisation: the ability to extract hyperslices to encapsulate concerns that were not separated in the original code. This makes hyperspaces especially useful for evolution of existing software.

1.4.3 Composition Filters

Composition Filters is developed by M. Akşit and L. Bergmans at the TRESE group, which is a part of the Department of Computer Science of the University of Twente, The Netherlands. The composition filters (CF) model predates aspect-oriented programming. It started out as an extension to the object-oriented model and evolved into an aspect-oriented model. The current implementation of CF is Compose⋆, which covers .NET, Java, and C.

One of the key elements of CF is the *message*. A message is the interaction between objects, for instance a method call. In object-oriented programming, the message is considered an abstract concept. In the implementations of CF, it is therefore necessary to reify the message. This *reified message* contains properties, like where it is send to and where it came from.

The concept of CF is that messages that enter and exit an object can be intercepted and manipulated, modifying the original flow of the message. To do so, a layer called the *interface part* is introduced in the CF model. This layer can have several properties. The interface part can be placed on an object, which behavior needs to be altered, and this object is referred to as *inner*.

```
hypermodule Pacman_Without_Debugging
  hyperslices: Feature.Kernel, Feature.Logging;
  relationships: mergeByName;
end hypermodule;
```

Listing 1.7: Defining a hypermodule

There are three key elements in CF: messages, filters, and superimposition. Messages are sent from one object to another, if there is an interface part placed on the receiver, then the message that is sent goes through the input filters. In the filters the message can be manipulated before it reaches the inner part, the message can even be sent to another object. How the message will be handled depends on the filter type. An output filter is similar to an input filter. The only difference is that it manipulates messages that originate from the inner part. The latest addition to CF is superimposition, which is used to specify which interfaces needs to be superimposed on which inner objects.

Chapter 2

Compose⋆

Compose⋆ is an implementation of the composition filters approach. There are three target environments: the .NET, Java, and C. This chapter is organized as follows, first the evolution of Composition Filters and its implementations are described, followed by an explanation of the Compose⋆ language and a demonstrating example. In the third section, the Compose⋆ architecture is explained, followed by a description of the features specific to Compose⋆.

2.1 Evolution of Composition Filters

Compose⋆ is the result of many years of research and experimentation. The following time line gives an overview of what has been done in the years before and during the Compose⋆ project.

1985 The first version of Sina is developed by Mehmet Akşit. This version of Sina contains a preliminary version of the composition filters concept called semantic networks. The semantic network construction serves as an extension to objects, such as classes, messages, or instances. These objects can be configured to form other objects such as classes from which instances can be created. The object manager takes care of synchronization and message processing of an object. The semantic network construction can express key concepts like delegation, reflection, and synchronization [43].

1987 Together with Anand Tripathi of the University of Minnesota the Sina language is further developed. The semantic network approach is replaced by declarative specifications and the interface predicate construct is added.

1991 The dispatch filter replaces the interface predicates, and the wait filter manages the synchronization functions of the object manager. Message reflection and real-time specifications are handled by the meta filter and the real-time filter [8].

1995 The Sina language with Composition Filters is implemented using Smalltalk [43]. The implementation supports most of the filter types. In the same year, a preprocessor providing C++ with support for Composition Filters is implemented [22].

1999 The composition filters language ComposeJ [61] is developed and implemented. The implementation consists of a preprocessor capable of translating composition filter specifications into the Java language.

2001 ConcernJ is implemented as part of a M. Sc. thesis [50]. ConcernJ adds the notion of superimposition to Composition Filters. This allows for reuse of the filter modules

	and facilitation of crosscutting concerns.
2003	The start of the Compose⋆ project, the project is described in further detail in this chapter.
2004	The first release of Compose⋆, based on .NET.
2005	The start of the Java port of Compose⋆.
2006	Porting Compose⋆ to C is started.

2.2 Composition Filters in Compose⋆

A Compose⋆ application consists of concerns that can be divided in three parts: filter module specifications, superimposition, and implementation. A filter module contains the filter logic to filter on incoming or outgoing messages on superimposed objects. Messages have a target, which is an object reference, and a selector, which is a method name. A superimposition part specifies which filter modules, annotations, conditions, and methods are superimposed on which objects. An implementation part contains the class implementation of a concern. How these parts are placed in a concern is shown in Listing 2.1.

The working of a filter module is depicted in Figure 2.1. A filter module can contain input and output filters. The difference between these two sets of filters is that the first is used to filter on incoming messages, while the second is used to filter on outgoing messages. The return of a method is not considered an outgoing message. A filter has three parts: a filter identifier, a filter type, and one or more filter elements. A filter element exists out of an optional condition part, a matching part, and a substitution part. These parts are shown below:

$$\overbrace{stalker\_filter}^{identifier} : \overbrace{Dispatch}^{filter\ type} = \{\overbrace{!pacmanIsEvil}^{condition\ part} =>$$
$$\overbrace{[*.getNextMove]}^{matching\ part}\ \overbrace{stalk\_strategy.getNextMove}^{substitution\ part}\ \}$$

```
concern {
  filtermodule {
    internals
    externals
    conditions
    inputfilters
    outputfilters
  }

  superimposition {
    selectors
    filtermodules
    annotations
    constraints
  }

  implementation
}
```

Listing 2.1: Abstract concern template

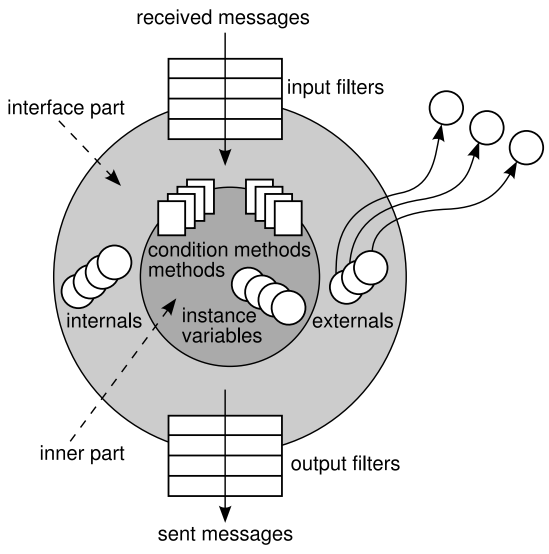


Figure 2.1: Components of the composition filters model

A filter identifier is a unique name for a filter in a filter module. Filters match when both the condition part and the matching part evaluate to true. In the demonstrated filter, every message where the selector is `getNextMove` matches. If an asterisk (`*`) is used in the target, every target will match. When the condition part and the matching part are true, the message is substituted with the values provided in the substitution part. How these values are substituted, and how the message continues, depends on the type of filter used. At the moment there are four basic filter types defined in Compose⋆. It is, however, possible to write custom filter types.

Dispatch If the message is accepted, it is dispatched to the specified target of the message, otherwise the message continues to the subsequent filter. This filter type can only be used for input filters;

Send If the message is accepted, it is sent to the specified target of the message, otherwise the message continues to the subsequent filter. This filter type can only be used for output filters;

Error If the filter rejects the message, it raises an exception, otherwise the message continues to the next filter in the set;

Meta If the message is accepted, the message is sent as a parameter of another meta message to an internal or external object, otherwise the message just continues to the next filter. The object that receives the meta message can observe and manipulate the message and can re-activate the execution of the message.

The identifier `pacmanIsEvil`, used in the condition part, must be declared in the conditions section of a filter module. Targets that are used in a filter can be declared as internal or external. An internal is an object that is unique for each instance of a filter module, while an external is an object that is shared between filter modules.

Filter modules are superimposed on classes using filter module binding, which specifies a selection of objects on the one side, and a filter module on the other side. The selection is specified in a selector definition. This selector definition uses predicates to select objects, such as `isClassWithNameInList`, `isNamespaceWithName`, and `namespaceHasClass`. In addition to filter modules, it is possible to bind conditions, methods, and annotations to classes using superimposition.

The last part of the concern is the implementation part, which can be used to define the behavior of a concern. For a logging concern, for example, we can define specific log functions and use them as internal.

2.3 Demonstrating Example

To illustrate the Compose⋆ toolset, this section introduces a *Pacman* example. The Pacman game is a classic arcade game in which the user, represented by pacman, moves in a maze to eat vitamins. Meanwhile, a number of ghosts try to catch and eat pacman. There are, however, four mega vitamins in the maze that make pacman evil. In its evil state, pacman can eat ghosts. A simple list of requirements for the Pacman game is briefly discussed here:

- The number of lives taken from pacman when eaten by a ghost;
- A game should end when pacman has no more lives;
- The score of a game should increase when pacman eats a vitamin or a ghost;
- A user should be able to use a keyboard to move pacman around the maze;
- Ghosts should know whether pacman is evil or not;
- Ghosts should know where pacman is located;
- Ghosts should, depending on the state of pacman, hunt or flee from pacman.

2.3.1 Initial Object-Oriented Design

Figure 2.2 shows an initial object-oriented design for the Pacman game. Note that this UML class diagram does not show the trivial accessors. The classes in this diagram are:

Game
This class encapsulates the control flow and controls the state of a game;

Ghost
This class is a representation of a ghost chasing pacman. Its main attribute is a property that indicates whether it is scared or not (depending on the evil state of pacman);

GhostView
This class is responsible for painting ghosts;

Glyph
This is the superclass of all mobile objects (pacman and ghosts). It contains common information like direction and speed;

Keyboard
This class accepts all keyboard input and makes it available to pacman;

Main
This is the entry point of a game;

Pacman
This is a representation of the user-controlled element in the game. Its main attribute is a property that indicates whether pacman is evil or not;

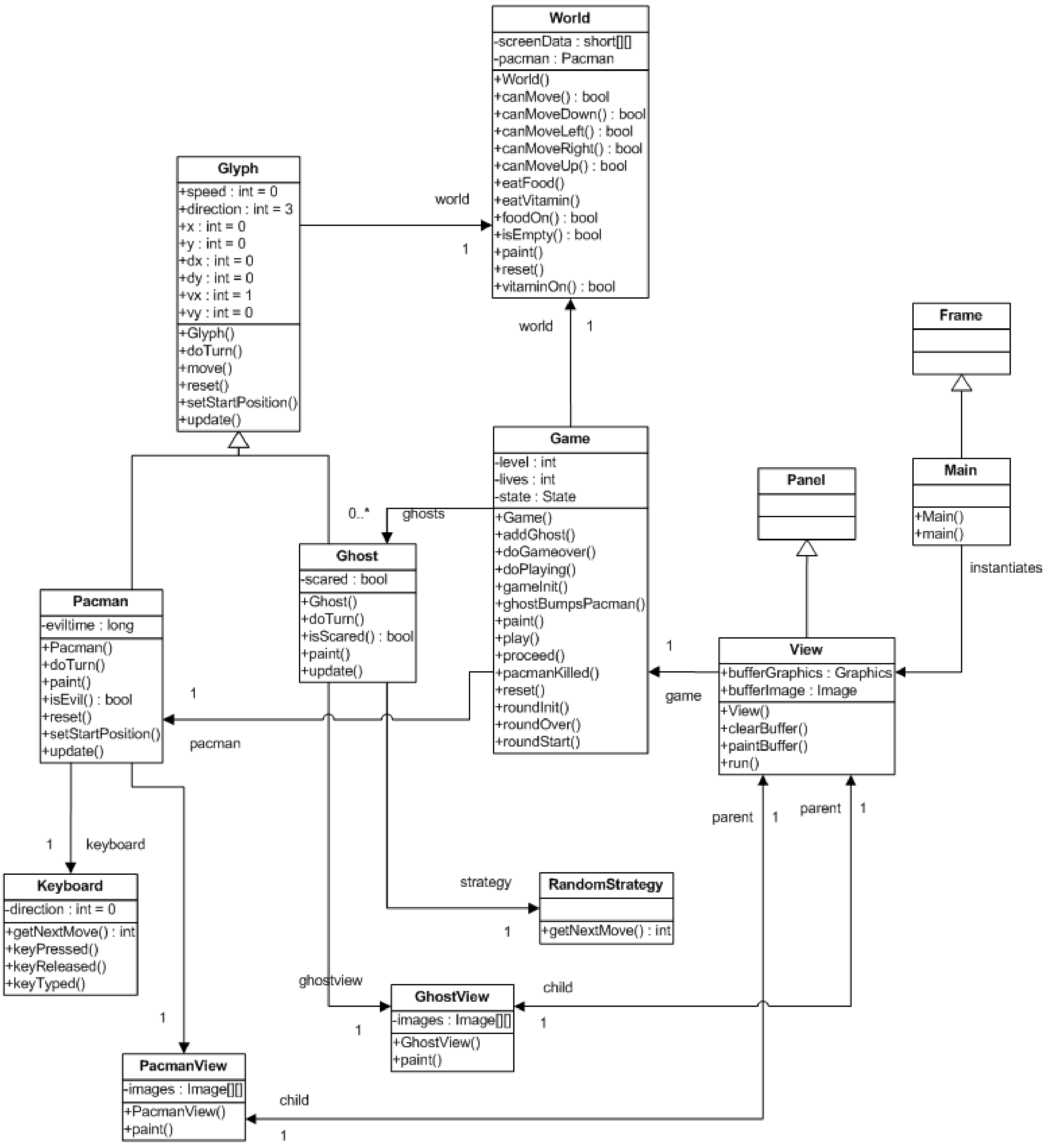


Figure 2.2: UML class diagram of the object-oriented Pacman game

PacmanView
: This class is responsible for painting pacman;

RandomStrategy
: By using this strategy, ghosts move in random directions;

View
: This class is responsible for painting a maze;

World
: This class has all the information about a maze. It knows where the vitamins, mega vitamins and most importantly the walls are. Every class derived from class `Glyph` checks whether movement in the desired direction is possible.

2.3.2 Completing the Pacman Example

The initial object-oriented design, described in the previous section, does not implement all the stated system requirements. The missing requirements are:

- The application does not maintain a score for the user;
- Ghosts move in random directions instead of chasing or fleeing from pacman.

In the next sections, we describe why and how to implement these requirements in the Compose⋆ language.

2.3.2.1 Implementation of Scoring

The first system requirement that we need to add to the existing Pacman game is scoring. This concern involves a number of events. First, the score should be set to zero when a game starts. Second, the score should be updated whenever pacman eats a vitamin, mega vitamin or ghost. Finally, the score itself has to be painted on the maze canvas to relay it back to the user. These events scatter over multiple classes: `Game` (initializing score), `World` (updating score), `Main` (painting score). Thus scoring is an example of a crosscutting concern.

To implement scoring in the Compose⋆ language, we divide the implementation into two parts. The first part is a Compose⋆ concern definition stating which filter modules to superimpose. Listing 2.2 shows an example Compose⋆ concern definition of scoring.

This concern definition is called `DynamicScoring` (line 1) and contains two parts. The first part is the declaration of a filter module called `dynamicscoring` (lines 2–11). This filter module contains one *meta filter* called `score_filter` (line 6). This filter intercepts five relevant calls and sends the message in a reified form to an instance of class `Score`. The final part of the concern definition is the superimposition part (lines 12–18). This part defines that the filter module `dynamicscoring` is to be superimposed on the classes `World`, `Game` and `Main`.

The final part of the scoring concern is the so-called *implementation part*. This part is defined by a class `Score`. Listing 2.3 shows an example implementation of class `Score`. Instances of this class receive the messages sent by `score_filter` and subsequently perform the events related to the scoring concern. In this way, all scoring events are encapsulated in one class and one Compose⋆ concern definition.

```
concern DynamicScoring in pacman {
  filtermodule dynamicscoring {
    externals
      score : pacman.Score = pacman.Score.instance();
    inputfilters
      score_filter : Meta = {[*.eatFood] score.eatFood,
                             [*.eatGhost] score.eatGhost,
                             [*.eatVitamin] score.eatVitamin,
                             [*.gameInit] score.initScore,
                             [*.setForeground] score.setupLabel}
  }
  superimposition {
    selectors
      scoring = { C | isClassWithNameInList(C, ['pacman.World',
                                 'pacman.Game', 'pacman.Main']) };
    filtermodules
      scoring <- dynamicscoring;
  }
}
```

Listing 2.2: `DynamicScoring` concern in Compose⋆

2.3.2.2 Implementation of Dynamic Strategy

The last system requirement that we need to implement is the dynamic strategy of ghosts. This means that a ghost should, depending on the state of pacman, hunt or flee from pacman. We can implement this concern by using the strategy design pattern. However, in this way, we need to modify the existing code. This is not the case when we use Compose⋆ *dispatch filters*. Listing 2.4 demonstrates this.

This concern uses *dispatch* filters to intercept calls to method `RandomStrategy.getNextMove` and redirect them to either `StalkerStrategy.getNextMove` or `FleeStrategy.getNextMove`. If pacman is not evil, the intercepted call matches the first filter, which dispatches the intercepted call to method `StalkerStrategy.getNextMove` (line 9). Otherwise, the intercepted call matches the second filter, which dispatches the intercepted call to method `FleeStrategy.getNextMove` (line 11).

2.4 Compose⋆ Architecture

An overview of the Compose⋆ architecture is illustrated in Figure 2.3. The Compose⋆ architecture can be divided in four layers [45]: IDE, compile time, adaptation, and runtime.

2.4.1 Integrated Development Environment

Some of the purposes of the Integrated Development Environment (IDE) layer are to interface with the native IDE and to create a build configuration. In the build configuration it is specified which source files and settings are required to build a Compose⋆ application. After creating the build configuration, the compile time is started.

The creation of a build configuration can be done manually or by using a plug-in. Examples

```java
import Composestar.Runtime.FLIRT.message.*;
import java.awt.*;

public class Score
{
  private int score = -100;
  private static Score theScore = null;
  private Label label = new java.awt.Label("Score: 0");

  private Score() {}

  public static Score instance() {
    if(theScore == null) {
      theScore = new Score();
    }
    return theScore;
  }

  public void initScore(ReifiedMessage rm) {
    this.score = 0;
    label.setText("Score: "+score);
  }

  public void eatGhost(ReifiedMessage rm) {
    score += 25;
    label.setText("Score: "+score);
  }

  public void eatVitamin(ReifiedMessage rm) {
    score += 15;
    label.setText("Score: "+score);
  }

  public void eatFood(ReifiedMessage rm) {
    score += 5;
    label.setText("Score: "+score);
  }

  public void setupLabel(ReifiedMessage rm) {
    rm.proceed();
    label = new Label("Score: 0");
    label.setSize(15*View.BLOCKSIZE+20,15*View.BLOCKSIZE);
    Main main = (Main)Composestar.Runtime.FLIRT.message.MessageInfo
                             .getMessageInfo().getTarget();
    main.add(label,BorderLayout.SOUTH);
  }
}
```

Listing 2.3: Implementation of class `Score`

```
concern DynamicStrategy in pacman {
  filtermodule dynamicstrategy {
    internals
      stalk_strategy : pacman.Strategies.StalkerStrategy;
      flee_strategy : pacman.Strategies.FleeStrategy;
    conditions
      pacmanIsEvil : pacman.Pacman.isEvil();
    inputfilters
      stalker_filter : Dispatch = {!pacmanIsEvil =>
                            [*.getNextMove] stalk_strategy.getNextMove};
      flee_filter : Dispatch = {
                            [*.getNextMove] flee_strategy.getNextMove}
  }
  superimposition {
    selectors
      random = { C | isClassWithName(C,
                            'pacman.Strategies.RandomStrategy') };
    filtermodules
      random <- dynamicstrategy;
  }
}
```

Listing 2.4: `DynamicStrategy` concern in Compose⋆

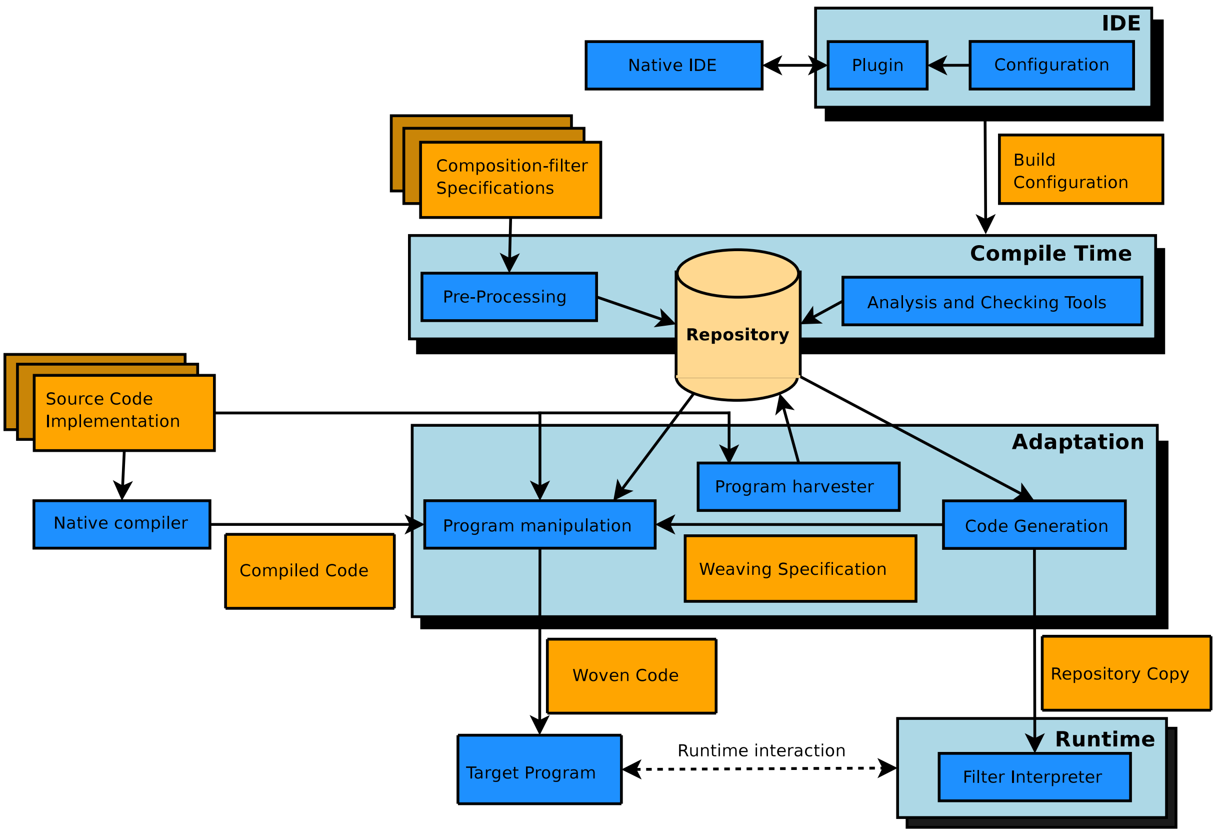


Figure 2.3: Overview of the Compose⋆ architecture

of these plug-ins are the Visual Studio add-in for Compose⋆/.NET and the Eclipse plug-in for Compose⋆/J and Compose⋆/C.

2.4.2 Compile Time

The compile time layer is platform independent and reasons about the correctness of the composition filter implementation with respect to the program which allows the target program to be build by the adaptation.

The compile time 'pre-processes' the composition filter specifications by parsing the specification, resolving the references, and checking its consistency. To provide an extensible architecture to facilitate this process a blackboard architecture is chosen. This means that the compile time uses a general knowledgebase that is called the 'repository'. This knowledgebase contains the structure and metadata of the program which different modules can execute their activities on. Examples of modules within analysis and validation are the three modules SANE, LOLA and FILTH. These three modules are responsible for (some) of the analysis and validation of the super imposition and its selectors.

2.4.3 Adaptation

The adaptation layer consists of the program manipulation, harvester, and code generator. These components connect the platform independent compile time to the target platform. The harvester is responsible for gathering the structure and the annotations within the source program and adding this information to the knowledgebase. The code generation generates a reduced copy of the knowledgebase and the weaving specification. This weaving specification is then used by the weaver contained by the program manipulation to weave in the calls to the runtime into the target program. The result of the adaptation is the target program that interfaces with the runtime.

2.4.4 Runtime

The runtime layer is responsible for executing the concern code at the join points. It is activated at the join points by function calls that are woven in by the weaver. A reduced copy of the knowledgebase containing the necessary information for filter evaluation and execution is enclosed with the runtime. When the function is filtered the filter is evaluated. Depending on if the condition part evaluates to true, and the matching part matches, the accept or reject behavior of the filter is executed. The runtime also facilitates the debugging of the composition filter implementations.

2.5 Platforms

The composition filters concept of Compose⋆ can be applied to any programming language, given that certain assumptions are met. Currently, Compose⋆ supports three platforms: .NET, Java and C. For each platform, different tools are used for compilation and weaving. They all share the same platform independent compile-time.

Compose⋆/.NET targets the .NET platform and is the oldest implementation of Compose⋆. Its weaver operates on CIL byte code. Compose⋆/.NET is programming language independent as long as the programming language can be compiled to CIL code. An add-in for Visual Studio is provided for ease of development. Compose⋆/J targets the Java platform and provides a plug-in for integration with Eclipse. Compose⋆/C contains support for the C programming language. The implementation is different from the Java and .NET counterparts, because it does not have a run-time environment. The filter logic is woven directly in the source code. Because the language C is not based on objects, filters are woven on functions based on membership of sets of functions. Like the Java platform, Compose⋆/C provides a plug-in for Eclipse.

2.6 Features Specific to Compose⋆

The Composition Filters approach uses a restricted (pattern matching) language to define filters. This language makes it possible to reason about the semantics of the concern. Compose⋆ offers three features that use this possibility, which originate in more control and correctness over an application under construction. These features are:

Ordering of filter modules
It is possible to specify how the superimposition of filter modules should be ordered. Ordering constraints can be specified in a fixed, conditional, or partial manner. A fixed ordering can be calculated exactly, whereas a conditional ordering is dependent on the result of filter execution and therefore evaluated at runtime. When there are multiple valid orderings of filtermodules on a join point, partial ordering constraints can be applied to reduce this number. These constraints can be declared in the concern definition;

Filter consistency checking
When superimposition is applied, Compose⋆ is able to detect if the ordering and conjunction of filters creates a conflict. For example, imagine a set of filters where the first filter only evaluates method *m* and another filter only evaluates methods *a* and *b*. In this case the latter filter is only reached with method *m*; this is consequently rejected and as a result the superimposition may never be executed. There are different scenarios that lead to these kinds of problems, e.g., conditions that exclude each other;

Reason about semantic problems
When multiple pieces of advice are added to the same join point, Compose⋆ can reason about problems that may occur. An example of such a conflict is the situation where a real-time filter is followed by a wait filter. Because the wait filter can wait indefinitely, the real-time property imposed by the real-time filter may be violated.

The above mentioned conflict analyzers all work on the assumption that the behavior of every filter is well-defined. This is not the case for the meta filter, its user-undefined, and therefore unpredictable, behavior poses a problem to the analysis tools.

Furthermore, Compose⋆ is extended with features that enhance the usability. These features are briefly described below:

Integrated Development Environment support
The Compose⋆ implementations all have a IDE plug-in; Compose⋆/.NET for Visual Studio, Compose⋆/J and Compose⋆/C for Eclipse;

Debugging support
The debugger shows the flow of messages through the filters. It is possible to place breakpoints to view the state of the filters;

Incremental building process
Incremental rebuilding re-uses the compilation results of previous buildings to safe compilation time

Some language properties of Compose⋆ can also be seen as features, being:

Language independent concerns
A Compose⋆ concern can be used for all the Compose⋆ platforms, because the composition filters approach is language independent;

Reusable concerns
The concerns are easy to reuse, through the dynamic filter modules and the selector language;

Expressive selector language
Program elements of an implementation language can be used to select a set of objects to superimpose on;

Support for annotations
Using the selector, annotations can be woven at program elements. At the moment annotations can be used for superimposition.

Nothing is a waste of time if you use the experience wisely.
– Auguste Rodin

Chapter 3

Problem Statement

To decrease compilation time, modern compilers support *incremental compilation*. Incremental compilers produce the same results as non-incremental compilers, but (re)compile only what is necessary in order to complete compilation as fast as possible. This chapter first describes several OOP and AOP incremental compilers and identifies the problems these compilers overcome. Then it introduces the non-incremental compilation process in Compose⋆ and finally, it describes the motivation for this thesis.

3.1 Background

In the OOP world, incremental compilation is well established. Numerous OOP incremental compilers compete with each other. Some examples are: Jikes [37], Eclipse compiler [17] and IBM's VisualAge C++ [31]. In this section, we first describe several OOP incremental compilation techniques and implementations of these techniques. Thereafter, we discuss the differences between incremental compilation in OOP and AOP. Finally, we describe existing AOP work on incremental compilation.

3.1.1 OOP Work on Incremental Compilation

An object-oriented program is generally coded across multiple source files. When a compiler compiles a program for the first time, it compiles *all* source files into binary format. Now imagine that a programmer makes a small change to a source file and subsequently asks a non-incremental compiler to recompile the program. In turn, a non-incremental compiler will recompile *all* source files. This strategy may result in redundant recompilations of source files, because the programmer's change might not affect all source files. Incremental compilers try to minimize the number of redundant recompilations. They achieve this by only recompiling the source files that are changed and the source files that are affected by the changes. The latter is the main issue for incremental compilers. *How does an incremental compiler know which source files are affected by a change in another source file?*

3.1.1.1 Make Tool

To answer the previous question, incremental compilers use various techniques. The *Make* program [20] implements the most simple (and most inefficient) one. *Make* is a UNIX utility that people use to manage source code compilation. It uses a *makefile* that contains dependency rules. These dependency rules define dependencies between source files. When a programmer changes one source file, *make* only compiles that source file and dependent source files. This approach has two major drawbacks:

File-level granularity of dependencies. The *make* tool only recognizes dependencies on file-level. This means that all types of changes made to a file result in recompilation of dependent files. Thus, this also applies when the programmer just adds some commentary. This is not ideal and obviously affects the number of redundant recompilations negatively.

Manual dependencies. The programmer needs to specify the dependencies in the *makefile* manually. This potentially leads to inconsistent compilation results and redundant recompilations. The first one may be the case when the programmer misses a dependency. The second one may be the case when the programmer defines too many or redundant dependency rules.

3.1.1.2 Jikes

To overcome the latter issue, some incremental compilers support automatic creation of a *makefile*. An example implementation is *Jikes* [37]. *Jikes* is a high performance Java compiler that performs dependency analysis for the programmer. This dependency analysis results in a *makefile* for each source file. To determine the dependency rules that make up a *makefile*, *Jikes* relies upon the class references found in the compiled source files (.class files).

3.1.1.3 Tichy's Smart Recompilation

However, *Jikes* still uses the file-level granularity of dependencies. Thus, small changes like adding commentary still may lead to redundant recompilations. To overcome this problem, Tichy [56] describes a finer grained solution for incremental compilation. This solution is called *"smart recompilation"*. The basic idea of this approach is as follows. After a program is modified, a change analysis will produce a *change set*. This *change set* consists of items that are added, changed or deleted. The *change set* is then intersected with the *reference set* of each compilation unit. The *reference set* of a compilation unit consists of items referenced by the compilation unit. A compilation unit is recompiled when the intersection of the *change set* and *reference set* is not empty. If the intersection is empty, the compilation unit is not recompiled. This approach avoids redundant recompilation after adding commentary because commentary is not supposed to be referenced by any compilation unit.

3.1.1.4 Eclipse Compiler

One successful implementation of Tichy's *"smart recompilation"* is Eclipse's Java compiler [17]. This compiler works as follows. To implement Tichy's *reference set*, the compiler maintains a *built state*. This *built state* includes a list of all types (classes and interfaces) that are referenced by each type in the *workspace*. Besides this built state, the compiler maintains a *resource delta*. This *resource delta* describes which files are added, removed and changed since last compilation. For removed files, the compiler deletes the corresponding .class files. Added and changed files are added to a queue of files. The compiler processes this queue as follows:

- Remove a file from the queue and compile it.
- Compare the resulting .class file with the .class file of the last compilation. See whether the type has some *structural changes*. Structural changes are changes that may affect the compilation of a referencing type. Examples are addition or removal of methods.
- If the type has structural changes, add the files of the referencing types to the queue.
- Update the built state according the new reference information of the compiled type.
- Repeat the above steps until the queue is empty.

This approach makes the Eclipse compiler fast. A programmer is namely likely to add just a few small changes. In addition, most of the time, these changes are not structural changes. Thus, this results in recompilation of only a small set of files.

3.1.1.5 IBM VisualAge C++

To conclude this section we shortly present an even more fine-grained solution. This solution is found in IBM's VisualAge C++ compiler [31]. This compiler keeps all declarations, definitions and their dependencies in a database called *CodeStore* [41, 44]. Whenever a source file changes, the compiler determines which of the declarations or definitions have been changed. Then the compiler only recompiles the changed declarations or definitions and the ones that are affected by the changes. Thus, the compiler only recompiles a minimal set of affected source code parts instead of whole source files.

3.1.2 AOP Work on Incremental Compilation

Compilation in AOP involves more complexity that compilation in OOP. More specifically, AOP compilers need to perform an additional operation called *weaving*. This *weaving* process integrates object-oriented components with declared aspects. Section 1.3.2 describes two widely-used weaving techniques: *source code weaving* and *byte-code weaving*.

3.1.2.1 Whole-program and Incremental Weaving

Most AOP implementations use a *whole-program weaving* technique [13] [14]. This means that even a single change to an aspect, leads to re-weaving of the entire program. Weaving is a costly operation. Therefore, some AOP implementations support *incremental weaving* [13]. Incremental weaving is to weaving as incremental compilation is to compilation [14]. This means that incremental weavers only weave (compiled) sources that are affected by changes to aspects.

3.1.2.2 Crosscutting Complicates Incremental Weaving

Supporting incremental weaving is complicated by the crosscutting nature of aspects because a single change to one aspect may lead to reweaving of an entire program [14]. For example, suppose that we add a logging concern that logs all method calls during the execution of a program. This will unavoidably affect the weaving of all (compiled) source files of the program, because the logging concern needs to be woven across the entire program.

3.1.2.3 Incremental Weavers

To conclude this section, we briefly present two AOP implementations that successfully implemented incremental weaving:

1. *ajc*. This is the original compiler for the AspectJ [7] language. The *ajc* compiler supports *incremental* byte-code weaving on a per-class basis. This means that the compiler minimizes the number of weave operations between classes and aspects in .class form. AspectJ recognizes the complication of aspects on incremental weaving by saying that the AspectJ tools are getting better at minimizing weaving, but to some degree, weaving is unavoidable due to the crosscutting semantics [2].

2. *Apostle*. Apostle is an aspect-oriented extension to the Smalltalk [51] programming language. Apostle uses *incremental* source-code weaving to weave Smalltalk source code and aspects together into equivalent pure Smalltalk results called *target models*. These models are compiled and installed by using a Smalltalk compiler. For further reading, [13] and [14] give a comprehensive description of the Apostle weaver implementation.

3.2 Non-incremental Compilation in Compose⋆

In this section, we describe the .NET Compose⋆ compilation process (before supporting incrementality). Figure 3.1 shows the control flow of the .NET Compose⋆ compilation process. In short, this process parses all user-provided project sources consisting of .NET sources and Compose⋆ aspects. Then it executes several analysis tools, and finally, it compiles and weaves the project sources into executable code. To realize this, the compilation process contains sixteen *compilation modules*. We call one of them *Master*. This compilation module serves as the main controller of the control flow. *Master* sequentially calls all other compilation modules, which in turn perform their own responsibilities and duties. All information produced by the compilation modules is stored into a central data store called *repository*. We briefly introduce the compilation modules below:

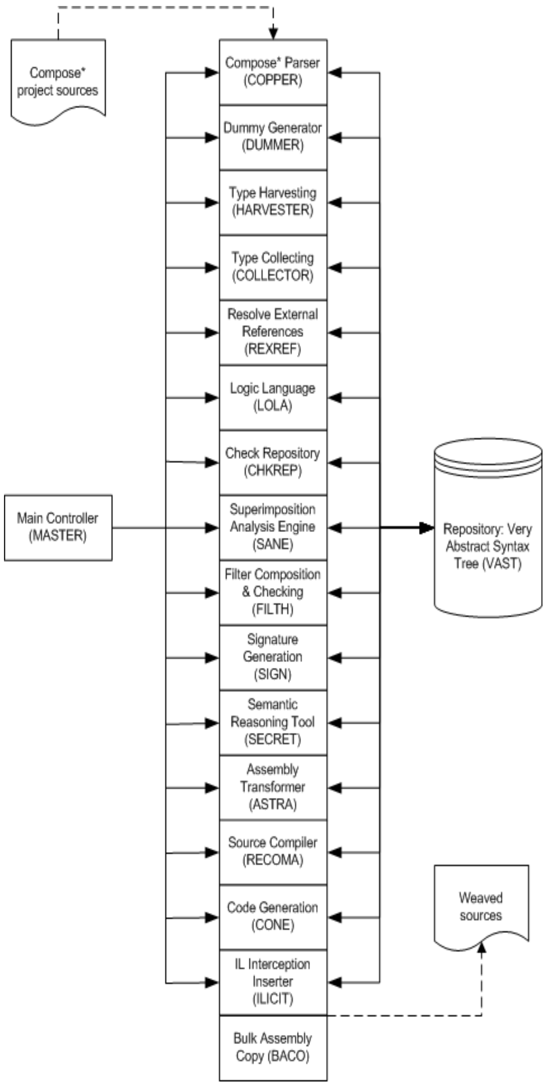

Compose* project sources
Compose* Parser (COPPER)
Dummy Generator (DUMMER)
Type Harvesting (HARVESTER)
Type Collecting (COLLECTOR)
Resolve External References (REXREF)
Logic Language (LOLA)
Check Repository (CHKREP)
Main Controller (MASTER)
Superimposition Analysis Engine (SANE)
Repository: Very Abstract Syntax Tree (VAST)
Filter Composition & Checking (FILTH)
Signature Generation (SIGN)
Semantic Reasoning Tool (SECRET)
Assembly Transformer (ASTRA)
Source Compiler (RECOMA)
Weaved sources
Code Generation (CONE)
IL Interception Inserter (ILICIT)
Bulk Assembly Copy (BACO)

Figure 3.1: Control flow of compilation

COPPER (Composestar Parser)
This is the parser of the source files that contain Compose⋆ concern definitions (files with .cps extension). It parses each concern source file and throws errors when it finds syntax errors. While parsing, it converts parsed concern data into Java objects and adds them to the repository.

DUMMER (Dummy Manager)
DUMMER transforms the .NET sources into *dummy sources* and compiles them by using a .NET compiler. A dummy source is a source with empty method blocks [27].

HARVESTER (Type Harvesting)
HARVESTER extracts meta-information (type and method signatures) from the input assemblies and writes this information to an XML file.

COLLECTOR (Type Collecting)
COLLECTOR parses the XML file produced by HARVESTER and stores the meta-information into the repository.

REXREF (Resolve External References)
Concerns may have both internal and external references to objects, methods and conditions. REXREF traverses the repository and resolves these references.

LOLA (Logic Language)
LOLA evaluates the selector queries expressed in prolog [25]. This results in sets of selected program elements per superimposition selector.

CHKREP (Check Repository)
CHKREP performs several sanity checks on the repository, e.g. the existence of unused references. This may result in errors and warnings presented to the user.

SANE (Superimposition Analysis Engine)
SANE imposes filtermodules on every program element selected by any superimposition selector.

FILTH (Filter Composition & Checking)
SANE produces information about where filtermodules are superimposed. However, it does not say anything about the order in which the filtermodules should be applied. FILTH therefore calculates all possible orderings of filtermodules. A user can put constraints on orderings by configuring the filter ordering specification file.

SIGN (Signature Generation)
Composition filters may grow or shrink the signature of a concern [27]. SIGN computes full signatures for all concerns and detects whether there are filters leading to ambiguous signatures.

SECRET (Semantic Reasoning Tool)
When multiple filtermodules are imposed on the same joinpoint, certain semantic conflicts may be introduced. SECRET aims to reason about these kinds of semantic conflicts. It performs a static analysis on the semantics of the filters and detects possible conflicts. The used model is, through the use of an XML input specification, completely user adaptable [16, 52].

ASTRA (Assembly Transformer)
ASTRA transforms the compiled dummy sources according to the full signatures calculated by SIGN.

RECOMA (Source Compiler)
RECOMA uses a .NET compiler to compile the .NET sources against the compiled dummy sources [27].

CONE (Code Generation)
CONE makes all repository data available at runtime by saving it to an XML file. CONE also creates a interception specification file containing instructions for the weaver.

ILICIT (Interception Inserter)

ILICIT is a .NET Intermediate Language (IL) weaver. It uses CONE's interception specification file to insert (weave) additional code in the .NET assemblies (at the execution joinpoints). The resulting weaved assemblies enforce the Compose⋆ runtime to execute the declared aspects [10].

BACO (Bulk Assembly Copy)

BACO copies all assemblies, created and referenced during compilation, to the output directory of the user-provided project. This ensures that Compose⋆ runtime has all resources to execute the compiled project correctly.

3.3 Motivation

The motivation for this thesis is to provide incremental compilation to Compose⋆. The current Compose⋆ version, described in the previous section, does not support incremental compilation. This means that whenever a programmer modifies a Compose⋆ project, the sixteen compilation modules perform their duties again. They ignore hereby all efforts and results of a previous compilation. While this non-incremental compilation is not a big problem for small Compose⋆ projects, it will give a huge burden for large Compose⋆ projects. Table 3.1 illustrates this.

Table 3.1: Compose⋆ compilation time for projects with different sizes

Size	Source files	Lines of Code	Classes	FilterModuleOrders	Time in seconds
Very Small	4	199	4	1	6.3
Small	21	2170	21	11	15.7
Medium	107	2872	107	124	58.2
Large	344	68459	472	731	391.5

Table 3.1 shows the compilation time for Compose⋆ projects with different sizes. The columns in this table are:

- *Size.* An indication of the project's size. The projects scale from "very small" to "large".
- *Source files.* The number of source files in the project.
- *Lines of Code.* The total lines of codes of all source files.
- *Classes.* The number of classes declared in the source files.
- *FilterModuleOrders.* A *filtermoduleorder* is an ordering of superimposed filtermodules. When we superimpose two *filtermodules* on a class, the number of filtermodule orderings for that class is two (one ordering and the reverse ordering). When we superimpose *n filtermodules* on a class, the number of filtermodule orderings for that class is *n!*. The *FilterModuleOrders* column shows the total sum of filtermodule orderings of all classes in the project.
- *Time in seconds.* The number of seconds spent to compile all source files in the project.

The table shows us that a programmer has to wait for about six and a half minutes to recompile a large project. But, the programmer often wants to recompile after making only a few small changes to a program. Thus, when the compiler only recompiles what is necessary, the programmer might not wait that long (see compilation time for small projects). In other words, by using incremental compilation we might decrease compilation time of Compose⋆ projects.

3.4 Problem Summary and Conclusion

The current version of Compose⋆ contains sixteen compilation modules. Each of them has their own duties and responsibilities. However, all of them suffer from the same problem. They do not support any kind of incrementality. This means that they perform their operations repeatedly without taking advantage of the work already done in a previous compilation. This unavoidably results in compilations consisting of numerous redundant operations. These redundant operations slow down compilation. To speed up the compilation, we want to add incremental performance to the Compose⋆ compilation modules. The next chapter elaborates on this problem and compares two possible solution approaches.

Chapter 4

Solution Approaches

The previous chapter describes the problem of incremental compilation in Compose⋆. This chapter first elaborates on this problem by describing a model of a compilation module in Compose⋆. Then, based on this model, it proposes two possible solution approaches for incremental compilation in Compose⋆. Finally, it defines a set of comparison criteria, applies this set on both approaches and selects the most desirable approach.

4.1 Model of a Compilation Module in Compose⋆

The previous chapter concludes that incremental compilation in Compose⋆ involves minimization of redundant operations performed by compilation modules. But, what do we mean when we speak of redundant operations of a compilation module? To answer this question, we use a black box model of a compilation module. Figure 4.1 shows this model.

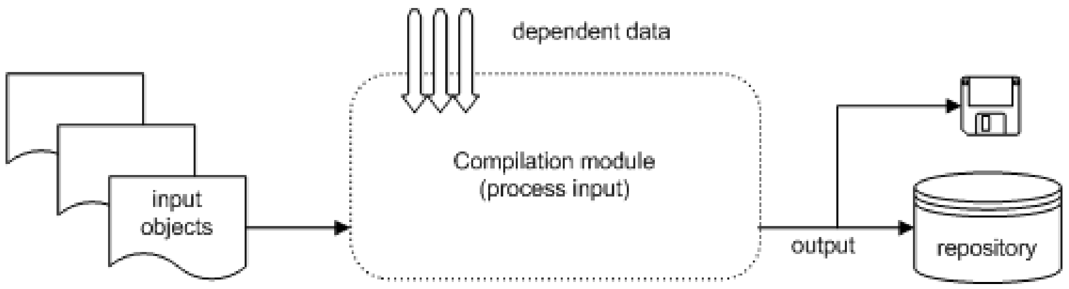


Figure 4.1: Model of a Compilation Module in Compose⋆

In the model of Figure 4.1, the rounded rectangle represents a *compilation module*. Compose⋆ contains sixteen of such compilation modules. See Section 3.2 for a brief description of these compilation modules. Each compilation module processes a set of *input objects*. To clarify this, we apply this to one of the sixteen compilation modules, called FILTH. This compilation module is responsible for calculating all possible orderings of the filtermodules superimposed on a concern. To fulfill this task, FILTH iterates over each concern available in the repository. Hence, the input objects of FILTH are all concerns available in the repository.

Each processing of an input object results in some output. This could either be data stored on disk or data inserted into a Compose⋆ repository. When we apply this to FILTH, we see that it calculates two objects called *"SingleOrder"* and *"FilterModuleOrders"*. It inserts both objects into the repository. The first object contains a first ordering of the filtermodules superimposed on the concern. The second object contains all possible orderings of the filtermodules superimposed on the concern. From this observation, we can conclude that the outputs of FILTH are the *"SingleOrder"* and the *"FilterModuleOrders"* objects.

The way a compilation module processes its input depends on a set of *dependent data* (represented by the block arrows). This set of dependent data is unique for each combination of input object and compilation module. Examples of dependent data are configuration files, project configurations and parts of the input object itself. For instance, the way FILTH processes its input (concerns) depends on two types of data. The first one is an XML ordering specification file, in which the user can put constraints on certain orderings. The second one is the names of all filtermodules superimposed on a concern. When you change one of these two data types for a concern, you might affect the filtermodule orderings calculated by FILTH for that concern. From the above model, we can now derive the following definition for a redundant operation of a compilation module.

A redundant operation of a compilation module is the processing of an input object that has already been processed in the previous compilation, and which results in exactly the same output as that previous processing. This operation is redundant because it repeats an operation performed in the previous compilation.

This definition raises several questions. For instance, how do you know whether an input object has already been processed in the previous compilation? When does a compilation module produce the same output as in the previous compilation? But most importantly, how can we minimize redundant operations without producing incorrect or unexpected compilation results? These questions need answers in order to establish incremental compilation in Compose⋆. The next two sections present two possible solution approaches to answer these questions.

4.2 Restoration in Seven Steps

The first solution approach takes the repository from a previous compilation as its starting point. During incremental compilation, all compilation modules adapt this repository. We address this model with the term *restoration*. Figure 4.2 shows a model of incremental restoration of the Compose⋆ repository.

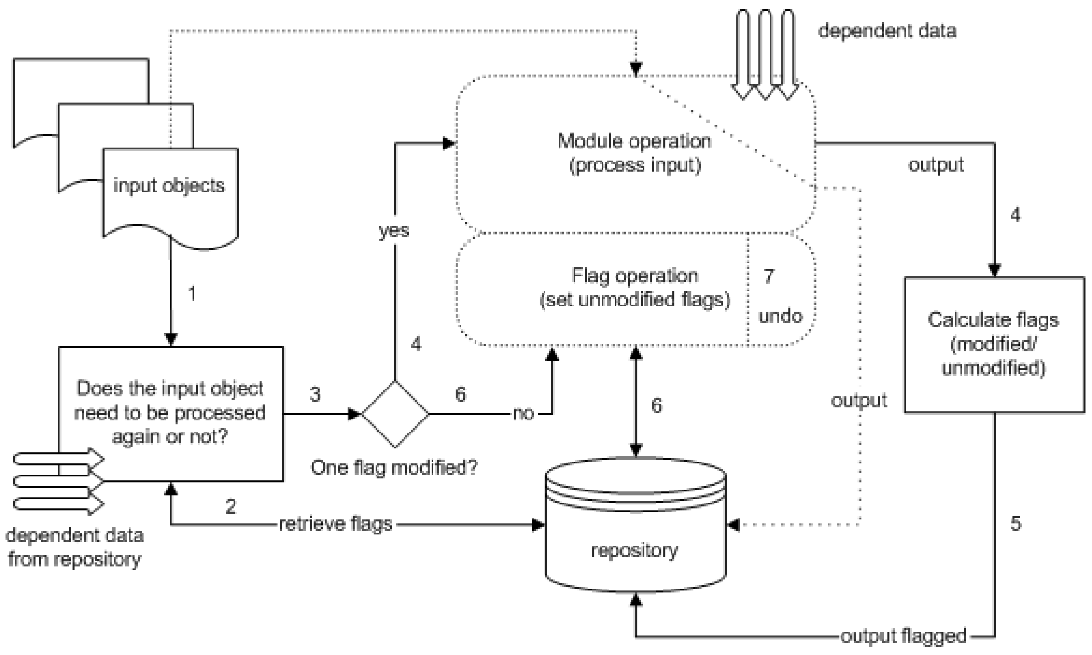

dependent data
input objects
Module operation (process input)
output
yes
Flag operation (set unmodified flags)
7
undo
4
Calculate flags (modified/ unmodified)
1
4
Does the input object need to be processed again or not?
3
6
no
6
output
One flag modified?
2
retrieve flags
dependent data from repository
repository
5
output flagged

Figure 4.2: Incremental restoration of the Compose⋆ repository

The dotted line in this model represents the non-incremental process of a compilation module described in Section 4.1. The solid line represents incremental restoration of the Compose⋆ repository. This incremental restoration model contains the following seven steps:

1. **Redirect input.** The first step in the restoration model is the redirection of the input of a compilation module to a new process. This means that the compilation module no longer processes its input directly. For each redirected input, the restoration model continues with step two.

2. **Retrieve set of flags.** As we have seen in Section 4.1, each input object has a unique set of dependent data. The restoration model supposes that each of these dependent data has a flag either *modified* or *unmodified*. These flags tell whether certain data differs from the previous compilation or not. At the start of the compilation, all flags are set to *modified*. In this second step, we collect the flags of all dependent data of the redirected input object.

3. **Check set of flags.** The third step in the restoration model is the most important one. In this step, we minimize the number of redundant operations of the compilation module by answering the following question for each redirected input object:

"Does the redirected input object need to be processed again by the compilation module?"

To answer this question, we use the retrieved set of flags as follows. When all flags are set to *unmodified*, we know that the input object and its dependent data have not been modified since the previous compilation. This means that processing the redirected input object again, would result in a repeat of a processing completed in the previous compilation. So, processing the redirected input object again would result in exactly the same output as produced in the previous compilation. But, because we have kept these compilation results, we do not need to reprocess the redirected input object again. Instead, we can take advantage of the work done in the previous compilation by reusing the kept compilation results. To achieve this, we answer the above question with *"no"* and continue with step six. When one flag is set to *modified*, we do not know for sure whether we can find the correct output in the kept compilation results. Hence, we must reprocess the redirected input object again. So, we answer the above question with *"yes"* and continue with step four.

4. **Process input.** In this step, a compilation module processes a redirected input object in its regular way. The output of this process is intercepted. For each intercepted output, the restoration model continues with step five.

5. **Flag output.** Recall that at the start of the compilation, we flag each compilation result as *modified*. To be able to skip redundant operations, however, we rely on the existence of *"unmodified"* flags (see step three). Hence, we need to flag the intercepted output repository objects and files before they are stored into the repository or stored on disk. To achieve this we do the following. For each intercepted output, find a *matching repository object* or a *matching file*. By a matching repository object, we mean a repository object that has the same identifier as the intercepted output repository object. By a matching file, we mean a file that has the same absolute path as the intercepted output file. When you found a match, compare it with the intercepted output. While comparing, flag the equal parts to *unmodified* and the unequal parts to *modified*. Finally, overwrite the matching repository object or matching file.

6. **Reuse kept compilation results.** The sixth step in the restoration model is an optimization step. In this step, we search for the compilation results of the skipped input object and set the flags of the found objects from *modified* to *unmodified*. By doing this, we tell the restoration model that the compilation results of the skipped input object have not been modified since the previous compilation. In other words, we are reusing these compilation results. In this way, we minimize the number of *modified* flags. This increases the chance of skipping next operations. This latter can be directly derived from step three.

7. **Clean up repository.** In the final step of the restoration model, we clean up outdated repository data. By outdated repository data, we mean repository data that are produced in a previous compilation and which are no longer consistent with the current compilation. We need this cleaning because of the following two reasons. First, without it, the repository would further grow after each compilation. Second, the processing of outdated repository data could potentially lead to incorrect and unexpected compilation results. To remove outdated repository data, each compilation module has a new *undo operation*. We invoke this undo operation after a compilation module has processed all its input.

4.3 Rebuilding in Five Steps

The second solution approach considers the repository from a previous compilation as a backup to rebuild the Compose⋆ repository faster. We address this model with the term *rebuilding*. Figure 4.3 shows a model of incremental rebuilding of the Compose⋆ repository.

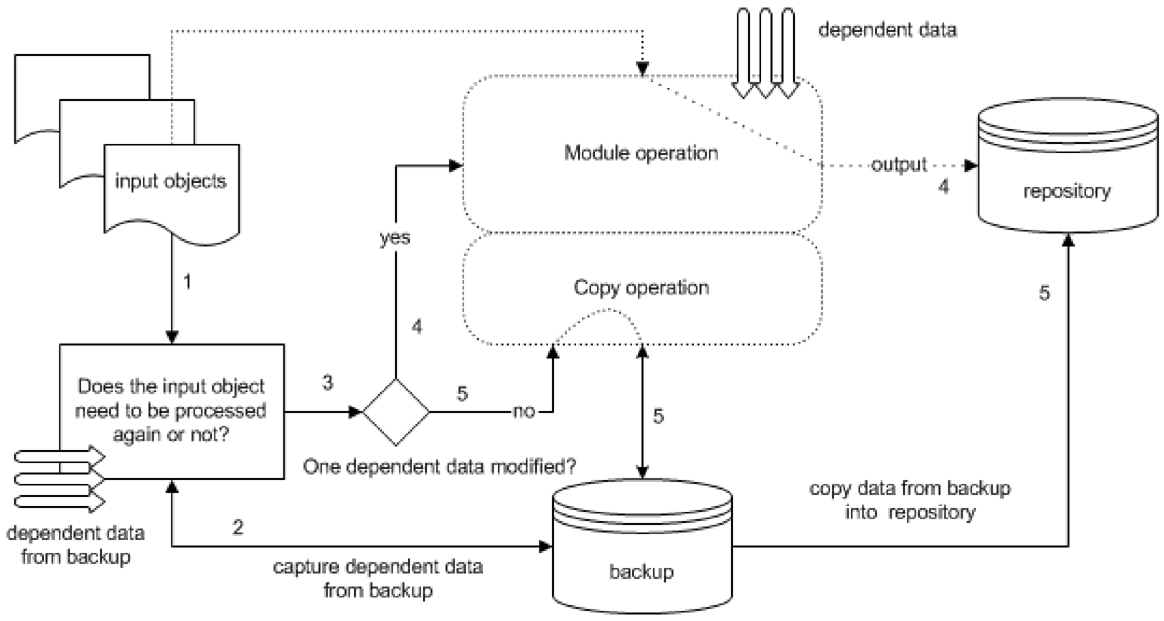

dependent data
input objects
Module operation
output
4
repository
yes
1
4
Copy operation
5
Does the input object need to be processed again or not?
3
5
no
5
One dependent data modified?
copy data from backup into repository
dependent data from backup
2
capture dependent data from backup
backup

Figure 4.3: Incremental rebuilding of the Compose⋆ repository

The dotted line in this model represents the non-incremental process of a compilation module described in Section 4.1. The solid line represents incremental rebuilding of the Compose⋆ repository. This incremental rebuilding model contains the following five steps:

1. **Redirect input.** The first step in the rebuilding model is the redirection of the input of a compilation module to a new process. This means that the compilation module no longer processes its input directly. For each redirected input, the rebuilding model continues with step two.

2. **Retrieve dependent data from repository and backup repository.** As we have seen in Section 4.1, each input object has a unique set of dependent data. In this second step, we retrieve these dependent data from the backup repository and the repository being built. After we have retrieved both sets of data, we continue with step three.

3. **Compare retrieved data sets.** The third step in the rebuilding model is the most important one. In this step, we minimize the number of redundant operations of the compilation module by answering the following question for each redirected input object:

   "Does the redirected input object need to be processed again by the compilation module?"

To answer this question, we compare the retrieved data sets of step two. When we do not find any mismatch, we know that the input object and its dependent data have not been modified since the previous compilation. This means that processing the redirected input object again, would result in a repeat of a processing completed in the previous compilation. In addition, processing the redirect input object again would result in exactly the same output as produced in the previous compilation. But, because we have kept these compilation results, we do not need to reprocess the redirected input object again. Instead, we can take advantage of the work done in the previous compilation by reusing the kept compilation results. To achieve this, we answer the above question with *"no"* and continue with step five. When we find at least one mismatch, we do not know for sure whether we can find the correct output in the kept compilation results. Hence, we must reprocess the redirect input object again. So, we answer the above question with *"yes"* and continue with step four.

4. **Process input.** In this step, a compilation module processes a redirected input object in its regular way. The output of this processing is stored on disk or added to the repository being built.

5. **Reuse kept compilation results.** In this step, we reuse the kept compilation results of the skipped input object. We achieve this by calling a new so-called *copy* operation of the compilation module. This copy operation it responsible for the following two tasks: finding the correct compilation results in the backup repository and copying these compilation results into the repository being built.

4.4 Comparison of Solution Approaches

Section 4.2 and Section 4.3 present *restoration* and *rebuilding* as solution approaches for incremental compilation in Compose⋆. To compare these approaches, we first define a set of comparison criteria. Then, we apply these criteria on both approaches and finally, we select the most desirable approach.

4.4.1 Comparison Criteria

To compare the solution approaches *restoration* (Section 4.2) and *rebuilding* (Section 4.3), we define the following comparison criteria:

Efficiency. The degree to which the incremental compiler can perform its designated functions with minimum consumption of resources like CPU and memory [5].

Simplicity. The degree to which the approach is straightforward and easy to understand [5].

Severity. The degree of impact that an implementation fault has on the development or operation of the incremental compiler [5].

Maintainability. The ease to which the implementation of the approach can be modified to correct faults, improve performance, or adapt to a changed environment [5].

4.4.2 Applying the Comparison Criteria

In this section, we apply the comparison criteria on the solution approaches *restoration* and *rebuilding*. Table 4.1 shows the degrees of each criteria applied on the approaches, with "++" being the highest degree, and "- -" the lowest degree.

Table 4.1: Comparison criteria applied to solution approaches

Approach	Efficiency	Simplicity	Severity	Maintainability
Restoration	+	+/-	-	- -
Rebuilding	+/-	++	+	+/-

4.4.2.1 Criteria applied to Restoration and Rebuilding

Efficiency. Restoration scores better on efficiency than rebuilding. The reason for this is twofold. First, restoration requires one instead of two repositories. Maintaining one repository involves consumption of resources like memory or disk (depending on the implementation). One extra repository unavoidably results in more consumption of these resources.
The second reason comes from the following estimation of the number of object comparisons in both approaches. First, consider the number of object comparisons in restoration. By using restoration, object comparison is required when a compilation module produces its output as normal (to flag it as unmodified or modified). These normal operations decrease when more flags are set to unmodified. The number of unmodified flags, in turn, likely increases when a programmer makes fewer changes to its program. From these observations, we conclude that the number of comparisons in restoration decreases when a programmer makes *fewer* changes to a program.
Now we consider the number of object comparisons in rebuilding. By using rebuilding, object comparison is required just before a regular operation of a compilation module. For each input object, we compare a set of dependent data with a set of data found in the backup repository. While comparing these sets, we can stop when we found one modification. On the other hand, when there are no modifications found, we will have to complete all object comparisons. From these observations, we conclude that the number of comparisons in rebuilding decreases when a programmer makes *more* changes to a program.
Now assume that a programmer is more likely to recompile after making a few small changes to a program rather than after making many changes. From this assumption and the above estimations, we conclude that restoration requires less object comparisons than rebuilding. In other words, restoration is potentially more efficient than rebuilding.

Simplicity. When it comes to simplicity, restoration scores worse than rebuilding. The main reason for this is the complexity of its flagging algorithm. Another reason for the lower degree of simplicity is the fact that the restoration model requires the addition of an undo operation to *all* compilation modules. Because, without these undo operations, the repository would further grow after each compilation and possibly contain outdated compilation results.

Severity. Restoration scores low on severity. The main reason for this is its ”All-or-nothing” character. This means that restoration requires implementation of incremental performance for each compilation module. Whether it is beneficial or not, we always need to add undo and flag operations. This implies that one implementation fault results in a loss of the whole incremental performance. Another reason for the low degree of severity is the fact that one compilation module could rely on the flags set by another compilation module. Hence, implementing a bad flagging algorithm for one compilation module could lead to a performance decrease of other compilation modules.
The above limitations do not apply to rebuilding. This is because, in rebuilding, the implementation and performance of a compilation module does not depend on other compilation modules. This makes is possible to shut down the incremental performance of a compilation module without losing incremental performance of other compilation modules. Thus, the impact of an implementation fault in rebuilding is less severe than in restoration.

Maintainability. The restoration model is hard to maintain because of its ”All-or-nothing” character. We need to enhance each compilation module with incremental performance. Whenever a compilation module is adapted, its incremental performance needs to be checked and possibly adapted as well. This is not the case in rebuilding, where incremental performance can be shut down and new compilation modules can be introduced without any incremental performance.

4.4.2.2 Conclusion

From the applied criteria, we draw the following conclusion. Restoration is the most efficient one but it has its costs. Compared to rebuilding, it is more complex, suffers more from implementation faults and is less maintainable. Therefore, we believe that rebuilding is the most desirable solution approach for incremental compilation in Compose⋆. In the next chapter, we compare several design alternatives for this solution approach.

The best way to escape from your problem is to solve it.
– Dr. Robert Anthony

Chapter 5

Design Alternatives Rebuilding Approach

Chapter 3 presented rebuilding as an approach for incremental compilation in Compose⋆. This chapter elaborates on this approach by first identifying the design decisions for this approach. After describing these decisions, it compares several design alternatives and chooses the most desirable ones. Finally, it summarizes our design solution for incremental compilation in Compose⋆.

5.1 Design Decisions

Development of the *rebuilding* (Section 4.3) model presents several design decisions for the developer. We classify these decisions into three categories:

Storage and Retrieval. The first design decisions relate to the storage and retrieval of compilation results. To produce the same results as a previous compilation, without recompiling, we need to preserve the previous compilation results. Section 4.1 tells us that the compilation results consist of data stored on disk and Java objects inserted into a repository. Since files are already preserved, we only need to find a way to preserve the repository data between compilations. The most commonly used term for this is *object persistence* [6].

Data Comparison. The second design decisions relate to the comparison of compilation data. To know what changed between two subsequent compilations, we need to have a mechanism to compare the compilation data of both compilations. Knowledge of the differences between compilations is indispensable, because it is a change that potentially leads to different compilation results.

Acquisition of Dependent Data. The last type of design decisions relate to the acquisition of dependent data of a compilation module. Our rebuilding approach relies on this data to tell whether a compilation module needs to process an input object again. The acquisition of this data should be accurate. We cannot afford to miss out any dependent data because of potential inconsistent compilation results. This is the case when we forget to check one modified dependent data for an input object, and consequently, falsely classify that input object as "not to be processed again". We also cannot afford to define too many dependencies, because this could lead to redundant dependency checking. The latter obviously affects the performance of the incremental compiler negatively.

In the next sections, we present design alternatives for each of the above design decisions and choose the most desirable ones.

5.2 Storage and Retrieval

The first design decision to make is how to make the Compose⋆ repository persistent. In the Java world, there are numerous persistence technologies (JOS [53], JDBC [32], JDO [35], Hibernate [26], Oracle TopLink [46]). In the next sections, we describe three persistence technologies: Java Serialization (JOS), Java Database Connectivity (JDBC) and Java Data Objects (JDO). Then, we define a set of criteria for successful usage in Compose⋆. Finally, we apply the set of criteria on the alternatives to make the best choice for Compose⋆.

5.2.1 Java Object Serialization (JOS)

Java Object Serialization [53] (JOS) is a well-established mechanism to persist objects. By using JOS, you can easily serialize object graphs to disk. The reverse operation, deserialization, is the process of reading data from disk and reconstructing the graph. Serialization and deserialization are accomplished with the `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` classes.

```
1  public void write(){ // serialize
2    ObjectOutputStream oos = new ObjectOutputStream(
3        new FileOutputStream("myfile.dat"));
4    oos.writeObject(new Date()); // write current date
5    oos.close(); // close stream
6  }
7
8  public void read(){ //deserialize
9    ObjectInputStream ois = new ObjectInputStream(
10       new FileInputStream("myfile.dat"));
11   Date d = (Date)ois.readObject(); // read date
12   ois.close(); // close stream
13 }
```

Listing 5.1: Java serialization and deserialization of current date

Listing 5.1 contains code for serialization and deserialization of the current date. The code contains two methods named `write` (lines 1– 6) and `read` (lines 8– 13). Within the `write` method, an instance of `java.io.ObjectOutputStream` is created to write objects to the file *"myfile.dat"*. Then, a new instance of class `java.util.Date` is instantiated. The date is written to the output stream by calling the `writeObject` method (line 4). After writing the date, the output stream is closed (line 5). A serialized object is read back by creating an instance of class `java.io.ObjectInputStream` (line 9) and invoking its `readObject` method (line 11). You need to cast read objects explicitly, as line 11 shows for the `java.util.Date` object. Again, we close the stream by calling the `close` method (line 12).

With just a few lines of code, you can write and read objects. The only restriction to this is that all objects to be persisted must implement the `java.io.Serializable` interface or inherit that implementation from its object hierarchy. Listing 5.2 shows how to accomplish this.

```
1 public class PersistentObject implements java.io.Serializable
2 {
3   //public void writeObject(){/*to support custom serialization*/}
4   //public void readObject(){/*to support custom deserialization*/}
5 }
```

Listing 5.2: Make objects ready for serialization

To make the class `PersistentObject` serializable, we just declare that the class implements the interface `java.io.Serializable` (line 1). The interface `java.io.Serializable` has two methods: `writeObject` and `readObject`. Finally, we may override these methods to support custom serialization and custom deserialization [24].

5.2.2 Java Database Connectivity (JDBC)

A second alternative for Java object persistence is Java Database Connectivity (JDBC). The JDBC API [32] provides access to database management systems (DBMS), which support the Structured Query Language (SQL). Further, JDBC allows you to create SQL statements that retrieve, store, update or delete Java objects in a database. Before you can use JDBC, you require a database driver [33] and add it to your classpath. Once added, you need to establish a connection with a DBMS. Listing 5.3 shows code that illustrates this.

```
1 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
2 String url = "jdbc:odbc:myDataSource";
3 Connection con = DriverManager.getConnection(url,"myLogin", "myPassword");
```

Listing 5.3: Establish a database connection with JDBC

First, you need to load the driver (line 1). `Class.forName` automatically creates an instance of a driver and registers it with the *DriverManager*. After loading the driver, you can connect the driver to a DBMS (lines 2– 3). To accomplish this, you need to set the connection URL. This URL depends on the type of driver you are using. When you are using the JDBC-ODBC Bridge driver, the URL starts with *jdbc:odbc*.

Once you have an active connection, you need to construct a JDBC statement to send SQL queries. You can send a SQL query by executing the appropriate execute method of the JDBC statement. For SELECT queries, the appropriate execute method is `executeQuery`. For queries that insert or modify data, the appropriate execute method is `executeUpdate`. Listing 5.4 demonstrates the use of the `executeQuery` method.

```
1 Statement stmt = con.createStatement();
2 ResultSet rs = stmt.executeQuery(
3             "SELECT Employee FROM personnel WHERE Employee.no=18");
4 rs.next();
5 Employee emp = (Employee)rs.getObject(1);
```

Listing 5.4: Retrieving Java Objects through JDBC statement

The example retrieves all employees with employee-number 18 from a table called *"personnel"*. First, it takes an instance of an active connection to create a `Statement` object (line 1). Then, we supply the JDBC Statement with an SQL query (line 2). The execution of this SQL query results in an instance of `ResultSet`. The `ResultSet` provides a cursor that you can move backwards and forwards to access rows in turn. By invoking the `next` method, we move the cursor to the first row (line 4). Finally, we obtain an `Employee` instance by calling the `getObject` method of the `ResultSet` instance (line 5).

When a DBMS receives a SQL query, it first parses the query to see if there are any syntax errors. Then, it needs to figure out the most efficient way to execute it. In other words, it creates a *query plan*. To avoid generating such a plan twice for the same queries, JDBC provides a `PreparedStatement`. Listing 5.5 demonstrates the use of prepared statements.

```
1 PreparedStatement pstmt = con.preparedStatement(
2       "SELECT Employee FROM personnel WHERE Employee.no=?");
3 pstmt.setInt(1,18);
4 pstmt.executeQuery();
```

Listing 5.5: Using JDBC *PreparedStatement*

Again, we want to retrieve all employees with employee-number 18. But, this time by using `PreparedStatement` instead of `Statement`. This gives us the possibility to parameterize the SQL statement. At line 1, we declare one `PreparedStatement` and supply it with a single parameterized SQL statement. The SQL statement expects one parameter, namely an employee number. We set this employee number by calling the `setInt` method (line 3). Finally, we execute the SQL statement by calling the `executeQuery` method (line 4).

Besides the use of 'prepared' statements, JDBC has some other features. To conclude our introduction of JDBC, we briefly summarize these features below:

Batch Processing. The ability to send multiple update SQL statements to process as a batch. This can be much more efficient than sending statements separately.

Transaction and Savepoints. The ability to use transactions and set savepoints to roll back transactions to.

Connection Pooling. The ability to cache and reuse connections.

Statement Pooling. The ability to cache and reuse statements.

5.2.3 Java Data Objects (JDO)

A third alternative for Java object persistence is Java Data Objects (JDO). The JDO API [35] is a specification to provide *transparent persistence* to developers of Java applications. Transparent persistence is the storage and retrieval of data with little or no work from the developer [57]. The JDO API primarily contains interfaces. So-called JDO implementations or JDO vendors implement these interfaces to comply with the JDO standard. These JDO vendors might either store Java objects in a relational database, object database or flat file. [38] lists a number of available commercial and non-commercial JDO implementations.

To persist objects, JDO requires all objects to implement the `javax.jdo.spi.PersistenceCapable` interface. Fortunately, unlike JOS, you do not need to add this interface by hand. Instead, each JDO vendor supports a so-called *enhancer*. An enhancer modifies the byte-code of a .class file based on XML meta-data. This meta-data specifies which classes and fields are persistent-capable. Figure 5.1 illustrates the enhancement process of a class `Employee`.

Just like JOS and JDBC, you need to set up a connection before you can persist objects. In JDO, connection to a data store is handled by an instance of `PersistenceManager`.

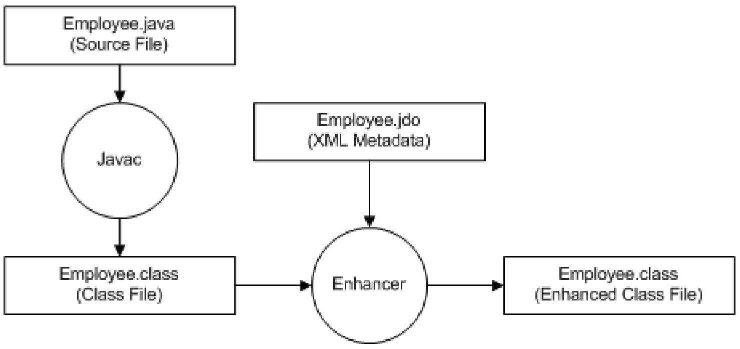


Figure 5.1: Illustration of the class enhancement process

`PersistenceManager` is the main interface of JDO. It is the factory for `Query` and `Transaction` instances, and has methods to manage a cache of `PersistenceCapable` instances. To obtain an instance of `PersistenceManager`, you can use `PersistenceManagerFactory`. Listing 5.6 shows example code that demonstrates this.

```
1 Properties props = new Properties();
2 props.put("javax.jdo.PersistenceManagerFactoryClass","myFactoryClass");
3 props.put("javax.jdo.option.ConnectionURL","myConnectionURL");
4 props.put("javax.jdo.option.ConnectionUserName","myUserName");
5 props.put("javax.jdo.option.ConnectionPassword","myPassword");
6 PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props);
7 PersistenceManager pm = pmf.getPersistenceManager();
```

Listing 5.6: JDO: obtain an instance of *PersistenceManager*

At lines 1– 5, we define some properties to configure the `PersistenceManagerFactory`. After configuring, we obtain an instance of `PersistenceManagerFactory` by calling the static `getPersistenceManagerFactory` method of class `JDOHelper` (line 6). Finally, we obtain an instance of `PersistenceManager` by calling the `getPersistenceManager` method of the `PersistenceManagerFactory` instance (line 7).

Once you have an instance of `PersistenceManager`, you can use it to store, retrieve, update or delete Java objects in your data store. The following listing demonstrates how to persist an object.

```
1 // pm is a PersistenceManager instance
2 Transaction tx = pm.currentTransaction();
3 tx.begin();
4 Employee emp = new Employee("John", 23, 40000);
5 pm.makePersistent(emp);
6 tx.commit();
```

Listing 5.7: Persist Java objects by using JDO

The example in Listing 5.7, persists a single instance of class `Employee` (constructed at line 4). To persist a Java object, you need to pass it to the `makePersistent` method of a `PersistenceManager` (line 5). To make use of JDO transactions, we place the call of the `makePersistent` method within a begin (line 3) and a commit (line 6) of a transaction. This ensures that the `Employee` is not persisted until we call the transaction's `commit` method. When you persist an instance, all referenced `PersistentCapable` instances are also automatically persisted. Thus, like JOS, you can persist entire object graphs by a single `makePersistent` call.

To conclude our introduction of JDO, we demonstrate how to retrieve JDO persisted objects. JDO supports two ways to retrieve persisted objects. The first one is via the `Extent` interface. By using this interface, you can retrieve all instances of a class and its subclasses. Listing 5.8 demonstrates the use of the `Extent` interface.

```
1 // pm is a PersistenceManager instance
2 Extent employees = pm.getExtent(Employee.class, true);
3 Iterator empItr = employees.iterator();
4 while(empItr.hasNext()){
5   Employee emp = (Employee)empItr.next();
6   System.out.println(emp.getName());
7 }
```

Listing 5.8: Retrieve Java objects by using the *Extent* interface

The example in Listing 5.8, prints the names of all persisted `Employees` to the console. To retrieve all persisted employees, we call the `getExtent` method of the `PersistenceManager` instance (line 2). To be able to iterate over the employees, we obtain the iterator of the returned `Extent` instance (line 3). Finally, we iterate over the employees and print their names to the console by calling `System.out.println` (lines 4– 7).

The second way to retrieve objects is to use the query language of JDO, called Java Database Objects Query Language (JDOQL). JDOQL is a query language whose syntax and expressions are largely based on Java. Listing 5.9 demonstrates the use of JDOQL.

```
1 // pm is a PersistenceManager instance
2 Extent employees = pm.getExtent(Employee.class, true);
3 String filter = "age == 23";
4 Query query = pm.newQuery(employees, filter);
5 Collection results = (Collection) query.execute();
```

Listing 5.9: Retrieve Java objects by using JDOQL

The example in Listing 5.9, retrieves all `Employees` of age 23. First, we declare the classes we are interested (line 2). In our case, we are looking for instances of class `Employee`. Then, we define a *query filter* to restrict our search to employees of age 23 (line 3). Once we have an `Extent` and a *query filter*, we can obtain an instance of class `Query`. We obtain a `Query` instance by calling the `newQuery` method of the `PersistenceManager` instance (line 4). Finally, we execute the `Query` instance by calling its execute method (line 5). As result, the executed `Query` returns an instance of class `Collection`. This collection contains all employees of age 23.

5.2.4 Comparison Criteria

To choose one of the three Java object persistence alternatives, we compare the alternatives on some comparison criteria. In this section, we shortly define our comparison criteria.

Ease of use. How much effort does it take, from both developer and user perspectives, to store and retrieve the Compose⋆ repository?

Portability. Do we need to rewrite code when we move the application from one platform to another?

Performance. How much time does it take to store the Compose⋆ repository? In addition, how fast can we retrieve an object or set of objects from the stored repository?

Scalability. How well does the mechanism perform as the Compose⋆ repository increases?

Note that whether the persistence mechanism provides a query language is not a criterion for us. We take this view because the Compose⋆ repository is designed to act like a database. Therefore, it already supports basic query operations like insertion, deletion and retrieval of objects by key. Further, we do not need concurrency control because of two reasons. First, there is only user (the programmer) that stores and retrieves the repository. Second, the compilation process is single threaded.

5.2.5 Applying the Criteria

In this section, we apply the comparison criteria on the three persistence alternatives. Table 5.1 shows the degrees of each criteria applied on the alternatives, with ”++” the highest degree, and ”-” the lowest degree. These degrees are not precise, but give merely an indication. Further, our conclusions are primarily based on the findings of [40] and [39].

Table 5.1: Comparison criteria applied to persistence mechanisms

Mechanism	Ease of Use	Portability	Performance	Scalability
JOS	++	+	+	-
JDBC	-	-	+/-	+/-
JDO	+	+/-	+	+/-

5.2.5.1 Criteria applied to JOS

Ease of use. From a developer perspective, JOS is easy to use. It has a simple API that makes it possible to store an entire object graph with just a few lines of code. From a user perspective, JOS is also very easy to use. Because it requires no further actions from the user except for file management. This is in contrast to JDBC and JDO, where the user might be required to install and manage a database.

Portability. Portability of JOS is high because it is a standard component of every Java Virtual Machine (JVM). However, the downside of JOS is that it does not guarantee serialization of objects across different Java versions. This means that, without adapting the serialization code, an object serialized in one Java version may not deserialize in another version.

Performance. JOS performs well for small objects compared to JDO and JDBC, because JOS does not need to perform actions for caching, transactions and concurrency control. Another benefit of JOS is that it does not require a connection to a database. On the downside, JOS does not support on-demand storage and retrieval of objects. This makes JOS unsuitable for applications that update serialized objects often.

Scalability. JOS scores low on scalability because it suffers from the ”big inhale and exhale” problem [19]. This problem refers to the fact that JOS only supports storage and retrieval of whole graphs and not partial graphs. Another downside of JOS's scalability is the potential `StackOverFlowEror`. This error could arise while serializing deeply nested object graphs, because JOS performs a depth-first search of the object graph. This means that JOS may visit the same objects twice. However, a JVM can only handle a limited number of objects before it throws a `StackOverFlowError`.

5.2.5.2 Criteria applied to JDBC

Ease of use. From a developer perspective, JDBC is more difficult than JOS and JDO because it requires knowledge of SQL queries. From a user perspective, JDBC is no more difficult than a JDO vendor implemented for databases, because both require installation and management of a database.

Portability. Code written for JDBC primarily contains SQL queries. SQL is a standard, so you would expect high portability. But, in practice, SQL queries are rarely portable across SQL implementations. [39].

Performance. The performance of JDBC is dominated by crossing the domain boundary between the Java environment and the SQL environment. Object-style navigation of data structures represented in SQL is extremely slow compared to the equivalent pointer-following operation of a JVM. The key to a good performance is to execute the majority of your JDBC application as SQL queries. Hereby, you minimize the number of round trips between the two environments and render the query optimization mechanism of the SQL server.

Scalability. In general, JDBC applications that do not create large numbers of objects that proxy data from the database scale well because they do not consume many JVM resources. In practice, however, JDBC applications are forced to explicitly cache data because of the high latency overhead of database round trips. This caching quickly becomes a source of complexity and errors [40]. Finally, JDBC scores better on scalability than JOS because, as a database solution, it does not suffer from the ”big inhale and exhale” problem.

5.2.5.3 Criteria applied to JDO

Ease of use. From a developer perspective, JDO is slightly more difficult than JOS. The developer needs to ”enhance” classes and maintain a meta data file written in XML. At a minimum, the meta data only specifies which classes are persistent. At a maximum, it also specifies the element type of collections, and whether, JDO should maintain an extent for a class. An extent is a mechanism to access all the instances of a class [39]. From a user perspective, JDO could be as easy as JOS. This is the case when the JDO vendor provides implementation for file stores. But, JDO vendors primarily provide implementation for relational and object databases. These implementations require installation and management of a database.

Portability. One objective of JDO is to allow an application to be portable across multiple JDO implementations. However, to port an application from one JDO implementation to another, both implementations need to support the same features. This is not always the case. Some features are optional and thus not supported by all implementations. To improve portability of JDO code, [36] describes some portability guidelines. These guidelines ensure that your JDO code is portable across JDO implementations.

Performance. For storage of small objects, JOS will generally perform better than JDO because there is simply less work to do. However, for larger objects, the differences reduce, because JOS suffers from scalability (see scalability applied on JOS). Retrieval of objects could be as fast as JOS, because of two features. Firstly, JDO has an automatic built-in caching mechanism. This means that it automatically adds all persisted objects to the cache of the `PersistenceManager`. This way, persisted objects stay in memory for fast retrieval. Note, however, that the caching performance differs between JDO implementations. The second reason, that might see JDO outperforming JOS in data retrieval, is that JDO allows retrieval of only the objects needed by the application. This is in contrast to JOS, which will always retrieve whole graphs of objects at once.

Scalability. Because of the caching mechanism mentioned above, JDO scores better on scalability than JOS. It does not suffer from the ”big inhale and exhale” problem as objects can be added and retrieved from the cache on demand.

5.2.6 Choice Motivation

Based on the following considerations we believe that JOS is the best solution for persistence of the Compose⋆ repository.

- JOS is the easiest way to achieve object persistence.
- We do not need concurrency control since there is always one user that stores and retrieves the repository and the compilation process is single threaded.
- JOS outperforms JDO and JDBC for small objects. The Compose⋆ repository currently scales from a couple of megabytes for small projects, to thirty megabytes for large projects. This is still considered as a ”small” object compared to enterprise applications (100-1000 megabytes), thus JOS should have the best performance.
- We do not need a powerful query language because the repository already supports basic queries like insertion, retrieval and deletion of objects by key.
- We can use custom serialization to control the class depth of the repository. This way, we can avoid the occurrence of a `StackOverFlowError`. The following example demonstrates this.

```
1  public abstract class FieldInfo extends ProgramElement
2  {
3    public String FieldTypeString;
4    private Type FieldType; //parent type
5
6    public Type fieldType()
7    {
8       if (this.FieldType == null)
9       {
10         TypeMap map = TypeMap.instance();
11         FieldType = map.getType( FieldTypeString );
12      }
13      return FieldType;
14   }
15 }
```

Listing 5.10: Recursion in Compose⋆ repository

  Listing 5.10 shows the code responsible for the deep recursion. Serialization of a `FieldInfo` object causes serialization of its parent object (line 4). This parent object contains the same field among others, which causes deep recursion. But, we do not need to serialize the field `FieldType` because of the existence of the `fieldType` method (lines 6– 14). By using custom serialization, we can choose which fields to serialize. Thus, we can control the class depth.

- JOS is not suitable for applications that update serialized data often, but we only need to update the serialized repository once at the end of the compilation.

5.2.7 Limitations of chosen alternative

When choosing JOS to persist the Compose⋆ repository, we recognize the following limitations:

- We must read the whole serialized repository once before we can use it. We cannot load parts of the repository data on-demand unless we store the repository data in separate files. Reading the whole repository has a negative impact on memory usage and performance.
- The scalability of JOS is poor. Currently, custom serialization and stack increases avoid the `StackOverFlowError` but that is not a guarantee for the future.
- Serialization creates copies of objects when they are written and read. This can break code based on hashcodes. This implies that we cannot compare objects based on hashcodes. Section 5.3 describes how this affects our way of data comparison.

5.3 Data Comparison

The second design decision for incremental rebuilding of the Compose⋆ repository is data comparison (see Section 5.1). More precise, we distinguish between two types of data comparison:

File Comparison. To tell whether a file changed since previous compilation.
Repository Entity Comparison. To tell whether a repository entity changed since previous compilation.

In the next sections, we discuss design alternatives for both types of data comparison.

5.3.1 File Comparison

In this section, we discuss two alternatives for file comparison: byte-by-byte and timestamp comparison. We compare the two alternatives on the following criteria:

Accuracy. How sure are we that a file changed since last compilation?
Performance. How fast can we conclude that a file changed since last compilation?

5.3.1.1 Byte-by-byte comparison

Byte-by-byte comparison takes two files, starts reading the bytes of both files and stops reading when it founds a mismatch in bytes. Since it compares all bytes, byte-by-byte is the safest way of file comparison. On the other hand, it scores low on performance. To use byte-by-byte comparison you need to have two files. For Compose⋆, this means that we need to copy each file used during a compilation cycle. Besides that, reading a file is a costly operation.

5.3.1.2 Timestamp comparison

Timestamp comparison is a solution based on the 'last-modified' timestamp of a file. To tell whether a file changed between compilation cycles, you only need to compare the 'last-modified' timestamp of the file with the completion timestamp of the last compilation. Therefore, timestamp comparison is much faster compared to byte-by-byte comparison. But, it has also two limitations that decrease accuracy:

1. Depending on the operating system, a copy of a file may not automatically have an updated timestamp. For instance, when you copy an old file on a Windows XP system, the copy has the same last-modified timestamp as the original file. Since we are only checking timestamps, we may wrongly classify a copied old file as *unmodified*.

2. If you add meaningless white spaces or save your file without making changes, you may update the timestamp of the file. This means that the file is classified as *modified* while the content has not changed. This is a wrong classification.

However, you can argue whether these scenarios are likely. We believe that these limitations do not outweigh the increase of performance. Based on this, we conclude that timestamp comparison is the most viable solution for file comparison in Compose⋆.

5.3.2 Repository Entities Comparison

The repository entities are Java objects. The main way to compare Java objects is to use the standard Java `equals` method. However, the JDK API [34] states one major contract between the `equals` and `hashCode` method:

Equal objects must produce the same hashcode!

But, hashcodes are not preserved when an object is serialized and subsequently deserialized by using JOS. Therefore, our choice for JOS breaks this contract. To repair the contract we should override the `equals` or `hashCode` method, but this is undesirable. Instead, we found a solution in Java reflection. Reflection gives us the possibility to configure the fields to use for object comparison manually. This is best explained by the following example:

Suppose that we have a class `Person` with three attributes: `name`, `age` and `nationality`. In addition, we have a configuration file that contains the following information:

- The fully qualified names of the classes to compare. In our case `Person`.
- The fields to use when comparing a class. In our example, we are only interested in the name of a person.

By using reflection, we can now compare two persons by name and ignore the two other attributes. Listing 5.11 shows example code that accomplishes this.

```java
import java.lang.reflect.Field;

public class Comparator {

  public HashMap fieldsByClasses = new HashMap();

  public boolean compare(Object a, Object b){
    Iterator fields = fieldsByClasses.get(a.getClass()).iterator();
    while(fields.hasNext()){
      String field = (String)fields.next();
      Object fielda = getField(a,field);
12       Object fieldb = getField(b,field);
13       if(!this.compare(fielda,fieldb)) return false;
14     }
15 
16     return true;
17   }
18 
19   public Object getField(Object obj, String field){
20     Field f = obj.getClass().getDeclaredField(field);
21     f.setAccessible(true); // to get access to private fields
22     return f.get(obj);
23   }
24 
25   public void parseConfigFile(){ /*code omitted*/ }
26 
27   public static void main(String[] args){
28     Comparator comp = new Comparator();
29     comp.parseConfigFile(args[0]);
30     Person p1 = new Person("John",21,"The Netherlands");
31     Person p2 = new Person("Mary",23,"Germany");
32     comp.compare(p1,p2);
33   }
34 }
```

Listing 5.11: Compare two persons by name using Java reflection and a configuration file

This code contains a class `Comparator` with one main method. This main method expects one argument, namely the path of a configuration file. The main method, firstly, creates an instance of class `Comparator` (line 28). Then, the information of the configuration file is retrieved by calling the `parseConfigFile` method (line 29). This method parses the configuration file and stores the configurations in a hashmap called *"fieldsByClasses"*. We have omitted the details of this method. After parsing the configuration file, the main method continues by creating two instances of class `Person` (line 30– 31). To compare the two instances of class `Person`, we call the `compare` method (line 32). This method retrieves the fields to compare and iterates over them (line 8). For each field, we collect the corresponding values (of the two compared objects). This is done by calling the `getField` method, which uses reflection. When there is a mismatch for a field, the comparator stops and returns `false` (line 13). Otherwise, it goes on to the next field. When all configured fields are equal, the comparator returns `true` (line 16).

Based on the above example, we constructed a solution model for comparing repository entities. This model is shown in Figure 5.2. To compare an object from the repository and its backup we first parse the configuration file. Then, by using reflection and the information from the configuration file, we collect the fields we want to compare. After collecting the fields, the second process collects the values of the fields. Once the values are collected, we can compare them. To compare the values we distinguish between four different types of values:

- Primitives. To compare primitives we use the standard '==' operator.
- Strings. To compare strings we use the standard `equals` method.
- Collections. Two collections are equal when sizes and all elements are equal. To compare two collections we first compare the sizes. After comparing the sizes, we compare the elements of the collections by iterating over them.
- Objects. In case the values are objects, we recursively start the comparing process.

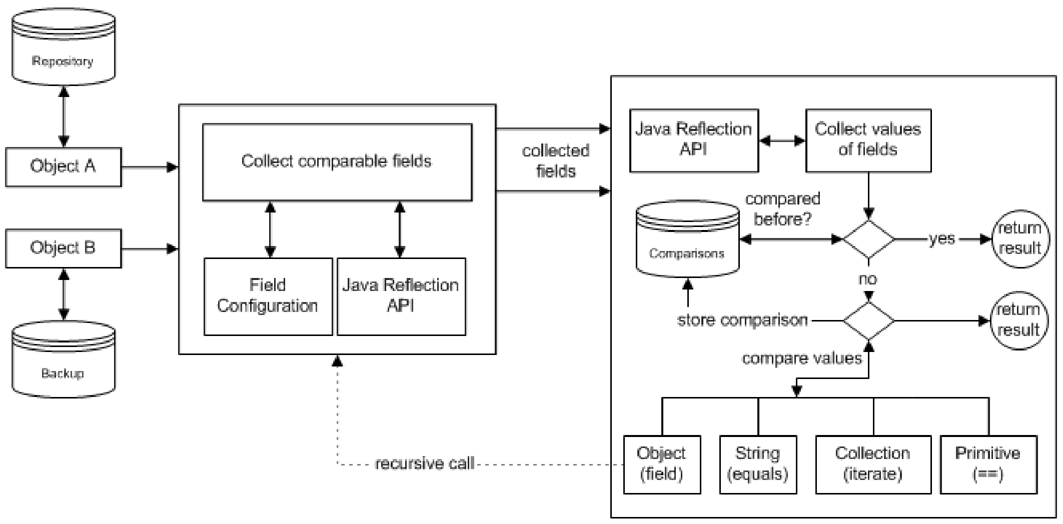


Figure 5.2: A recursive, generic way of comparing objects using reflection

Finally, as an optimization, we keep track of the comparison results. When we must compare two objects that we have compared before, we return the result of that previous comparison. This way, we avoid performing the same comparison more than once.

5.4 Acquisition of Dependent Data

The last design decision for incremental rebuilding of the Compose⋆ repository is how to acquire the dependent data of a compilation module (see Section 5.1). Since this data is subject to change, we have chosen to configure these data dependencies in an XML configuration file. Listing 5.12 presents an XML Document Type Definition (DTD) [4] of our dependency configuration file.

```
<!ELEMENT module (dependency)*>
  <!ATTLIST module
      name CDATA #REQUIRED>

<!ELEMENT dependency (dependencypath)>
  <!ATTLIST dependency
      name CDATA #REQUIRED
      type ("FILE" | "OBJECT")>

<!ELEMENT dependencypath (dependencynode)*>
  <!ATTLIST dependencypath EMPTY>

<!ELEMENT dependencynode EMPTY>
  <!ATTLIST dependencynode
      type ("CONFIG" | "DYNAMIC" | " FIELD" | "METHOD")
      value CDATA #REQUIRED>
```

Listing 5.12: XML DTD of a dependency configuration file for Compose⋆

In this XML configuration file, each compilation module has an entry (line 1). For each compilation module, we can configure a set of data dependencies. A data dependency has three attributes:

Name. The first one is a naming property (line 7). This attribute is used for identification.

Type. The second one refers to the type of dependency (line 8). We distinguish between two types of dependencies: a file dependency and an object dependency. This distinction is necessary when it comes to data comparison (see Section 5.3).

Path of dependency. The last attribute is the most important and difficult one. It describes a way to retrieve dependent data for any input object of the compilation module. A path of dependency nodes (line 10) represents the way of retrieval. Each node refers to a way of data retrieval. More specifically, each node expects a Java object as input and returns a Java object as output. The output of one node is passed on to the next node. Thus, by setting the input of the first node to a specific input object of the compilation module, and visiting the path of dependency nodes, we acquire the configured dependent data for the specified input object. To be able to configure the dependencies in different ways, we have chosen for the following four types of nodes (line 15):

- *Field node*. This node references a field of the input object. The output of this node is the value of the referenced field.
- *Config node*. This node references a project configuration. Project configurations are configurations that can be set by the user per Compose⋆ project. The output of this node is the value of the referenced project configuration.
- *Dynamic node*. Each repository entity has a map containing zero or more 'dynamic' objects. This node references a dynamic object 'attached' to the input object. The output of this node is the value of the referenced dynamic object.
- *Method node*. This is a reference to a method. By visiting this node, the specified method is invoked. The parameter of this method is the input of the node. The output of this node is the return value of the invoked method.

This design solution gives us multiple ways to define data dependencies for Compose⋆ compilation modules. It is our task to find the most suitable one.

5.5 Conclusion

This and the previous chapter presented in detail our solution for incremental compilation in Compose⋆. This solution involves persistence of the repository by using JOS, incremental rebuilding of the repository by replacing redundant operations by so-called copy operations, comparison of files and repository entities by using timestamp comparison and Java reflection, and configuration of the data dependencies of a compilation module by using an XML configuration file. To summarize, we here briefly outline a five-step procedure for enhancing a Compose⋆ compilation module with incremental performance:

1. Define the input, output and data dependencies of a compilation module. Each compilation module processes an input object or a set of input objects. Each input object processed by a compilation module results in some output. This output could be repository entities as well as files stored on disk. The way a compilation module processes its input objects depends on a set of data. This set of data dependencies is unique for each combination of compilation module and input object. Knowledge of input, output and data dependencies of a compilation module is indispensable for performing the next tasks.

2. Store and retrieve all repository objects produced and used by the compilation module by using Java's standard object serialization. Break potential deeply nested recursion by using custom serialization and custom deserialization. Ensure that each repository object produced and used by the compilation module implement the `java.io.Serializable` interface or inherit that implementation from its object hierarchy. Otherwise, the serialization process will throw a `java.io.NotSerializableException`.

3. Add a copy operation to a compilation module (see Section 4.3). This copy operation should copy the output belonging to a specific input object from the backup repository into the repository being built. Note that this copy operation should be faster than an original processing of an input object in order to safe compilation time. Further, this copy operation only applies to compilation modules that produce one or more repository objects and not files. The reason for this is that it is redundant to replace a file with a copy of that file.

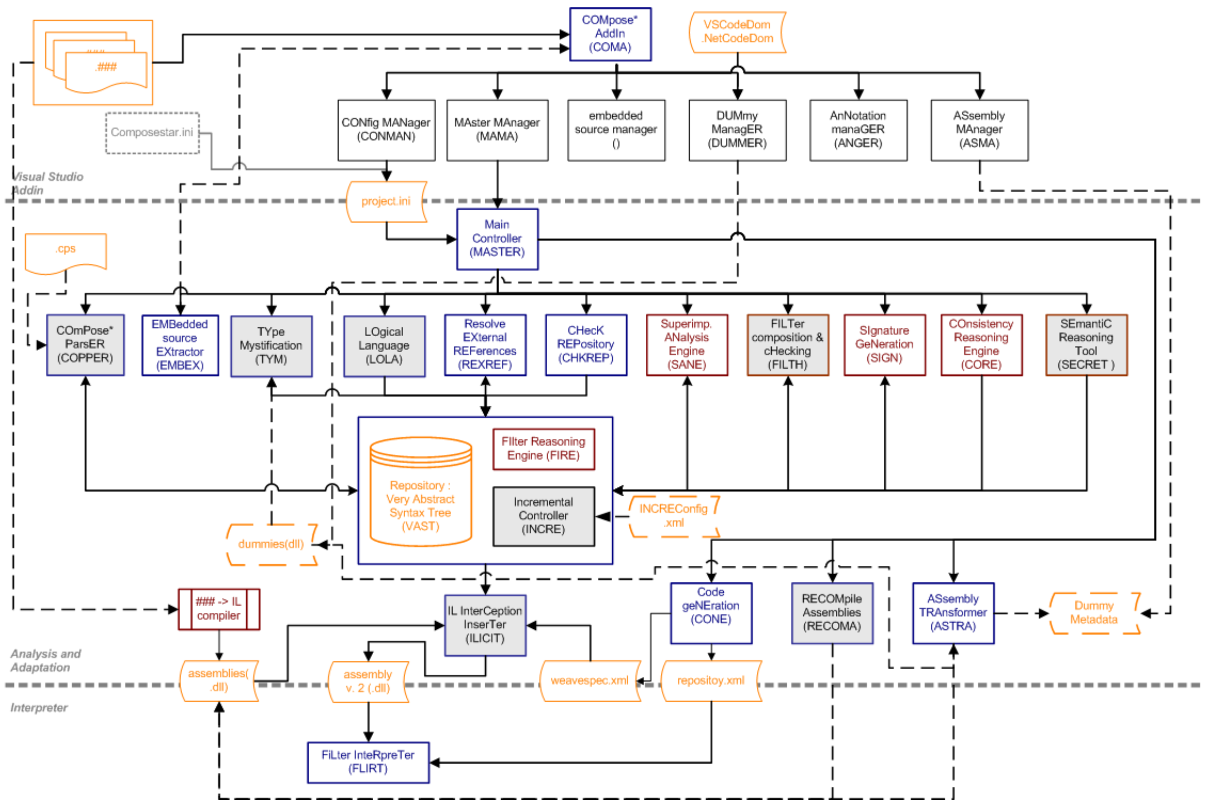
4. Configure the identified data dependencies of a compilation module by using a XML configuration file. This XML configuration file provides several XML tags to define data dependencies of a compilation module (see Section 5.4).

5. Define and configure the fields to use for repository entity comparison (see Section 5.3.2). For each compilation module, you can configure on which fields to compare two objects of a certain type. If you do not configure these fields, the comparator will compare two objects on all fields by default. Otherwise, it compares two objects on the configured fields only. This way, you can minimize the number of comparisons.

Chapter 6

Implementation

This chapter presents an implementation of the incremental compiler described in Chapter 4 and Chapter 5.



COMpose* AddIn (COMA)
VSCodeDom .NetCodeDom
###
Composestar.ini
CONfig MANager (CONMAN)
MAster MAnager (MAMA)
embedded source manager ()
DUMmy ManagER (DUMMER)
AnNotation manaGER (ANGER)
ASsembly MAnager (ASMA)
Visual Studio Addin
project.ini
Main Controller (MASTER)
.cps
COmPose* ParsER (COPPER)
EMBedded source EXtractor (EMBEX)
TYpe Mystification (TYM)
LOgical Language (LOLA)
Resolve EXternal REFerences (REXREF)
CHecK REPository (CHKREP)
Superimp. ANalysis Engine (SANE)
FILTer composition & cHecking (FILTH)
SIgnature GeNeration (SIGN)
COnsistency Reasoning Engine (CORE)
SEmantiC Reasoning Tool (SECRET)
Repository : Very Abstract Syntax Tree (VAST)
FIlter Reasoning Engine (FIRE)
Incremental Controller (INCRE)
INCREConfig .xml
dummies(dll)
### -> IL compiler
IL InterCeption InserTer (ILICIT)
Code geNEration (CONE)
RECOMpile Assemblies (RECOMA)
ASsembly TRAnsformer (ASTRA)
Dummy Metadata
Analysis and Adaptation
assemblies(.dll)
assembly v. 2 (.dll)
weavespec.xml
repositoy.xml
Interpreter
FiLter InteRpreTer (FLIRT)

Figure 6.1: Compose⋆ architecture with INCRE and adapted compilation modules highlighted

6.1 Adaptations to the Compose⋆ architecture

This section gives an overview of the adaptations made to the Compose⋆ architecture. Figure 6.1 shows the Compose⋆ architecture with the new or adapted components highlighted.

A new compilation module called `INCRE` manages incremental compilation. This compilation module is designed as a service for all other Compose⋆ compilation modules. The highlighted compilation modules use this new service. Figure 6.2 shows an UML static structure of `INCRE`.

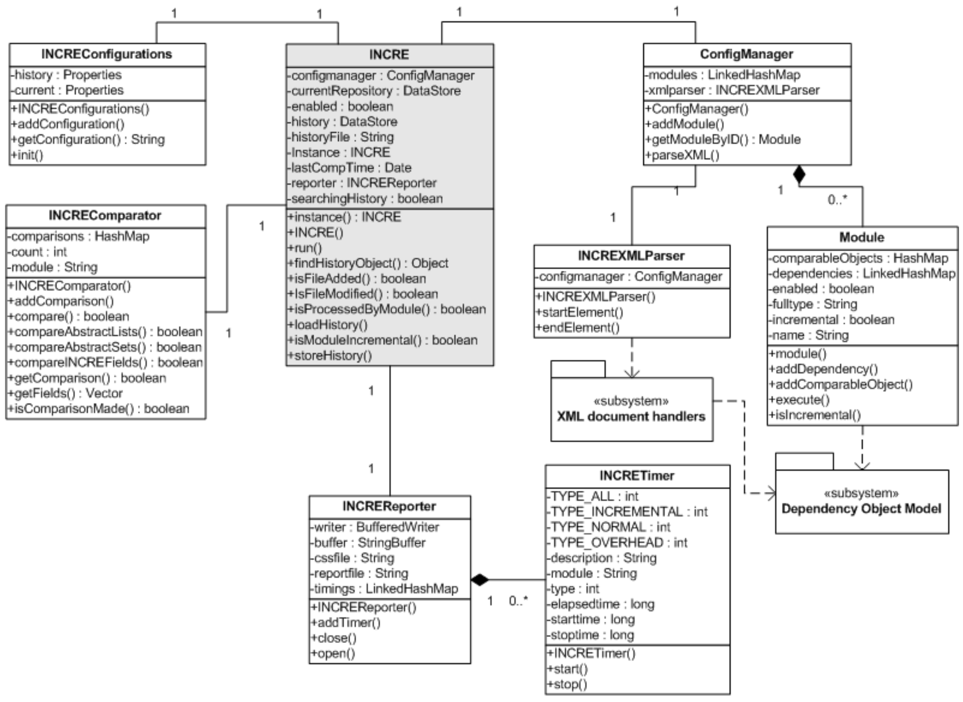


Figure 6.2: UML static structure of compilation module `INCRE`

The components in the INCRE package are:

INCRE This is the heart of the incremental compilation. It is responsible for the decision whether to reprocess input of a compilation module or not. It also stores and retrieves the compilation results.

INCREConfigurations This class contains the user-provided project configurations used in the previous and current compilation process.

INCREComparator This is the comparator mechanism discussed in Section 5.3. It compares data from current and previous compilations and keeps track of the completed comparisons. Section 6.3 discusses this mechanism in more detail.

ConfigManager This class keeps a list of all Compose⋆ compilation modules extracted from the XML configuration file. It is also responsible for setting up the XML parser.

INCREXMLParser This is the parser of the XML configuration file. The parser contains multiple XML document handlers. Section 6.2 describes the implementation of this parser.

Module This class represents a Compose⋆ compilation module extracted from the XML configuration file. Each compilation module has a set of data dependencies. These data dependencies are structured in an object model. Section 6.2 describes this object model.

INCRETimer This is a stopwatch for timing the different compilation processes.

INCREReporter This is the reporter of the profiling results. It generates timings reports in HTML format. Section 6.4 describes this reporter in more detail.

6.2 XML configuration file

Incremental compilation is configured with an XML configuration file called *"INCREConfig.xml"*. Appendix A summarizes the type of configurations. Appendix B shows an example configuration. The XML file contains information on all Compose⋆ compilation modules. An XML parser reads this information. Figure 6.3 shows the implementation of this XML parser.

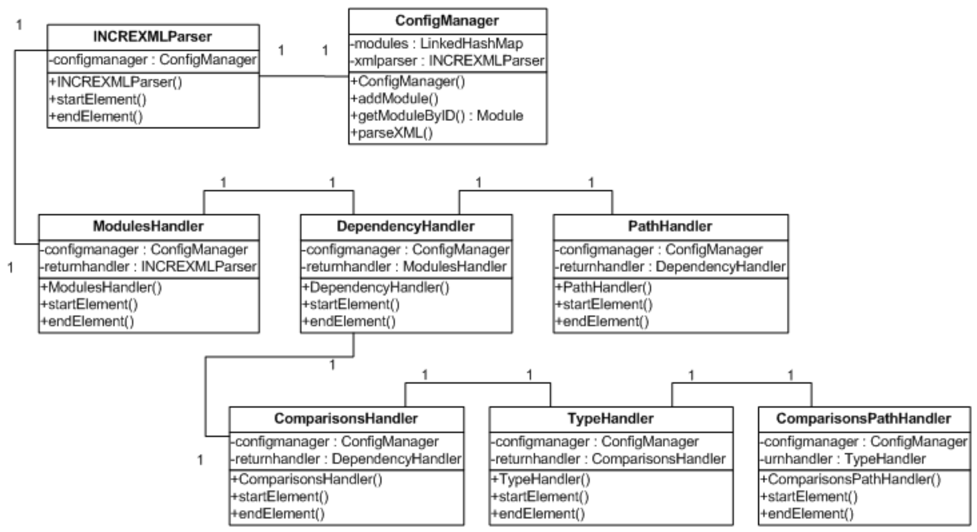


Figure 6.3: UML static structure of the parser of the XML configuration file

An instance of `INCREXMLParser` opens the XML file and parses it. It uses the standard Java SAX API [29]. This API is event based. Each XML parser that implements the SAX API generates events while parsing XML. Our XML parsers implement the events `startElement` and `endElement`. These events represent the start and end of tags. There is one handler for each type of tag. All information found by the parser is converted to objects. Figure 6.4 shows the object model of the XML configuration file.

In this figure, class `Module` represents a Compose⋆ compilation module. An instance of `ConfigManager` maintains a list of these compilation modules. Each compilation module has the following attributes:

- A *name* used for identification.
- A boolean *enabled*. Whether the compilation module is enabled or not.
- A boolean *incremental*. Whether the compilation module is incremental or not.
- The fully qualified type of the represented compilation module. This string is used to create and run the represented compilation module by using reflection.
- A mapping between types and fields. For each type you can define a list of fields to use for object comparison. Section 6.3 discusses object comparison in more detail.
- A map of dependencies. A dependency either represents a file (`FileDependency`) or a Java object (`ObjectDependency`). The value of a dependency is found with the `getDepObject()` method. This method visits a 'dependency path'. A dependency path is a collection of dependency nodes. Each node expects a Java object as input and returns a value either related to the input or not. The four types of nodes are (see also Section 5.4):
    1. `ConfigNode`, which returns the referenced project configuration.

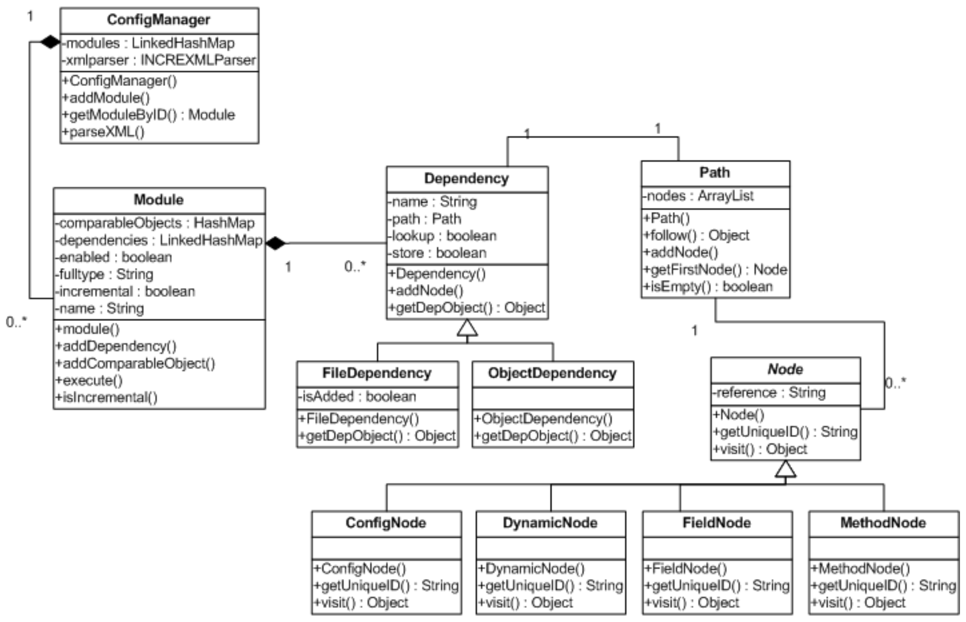


Figure 6.4: XML configuration file converted to an object model

2. `FieldNode`, which returns the referenced field of the input object.
3. `MethodNode`, which invokes the referenced method with the input object as first parameter and returns the return object of the invoked method.
4. `DynamicNode`, which returns the referenced dynamic object attached to the input object.

6.3 Implementation of the object comparator mechanism

The class `INCREComparator` implements the solution model for object comparison, described in Section 5.3.2. Figure 6.5 shows a simple UML sequence diagram of the interaction between `INCRE` and `INCREComparator`.

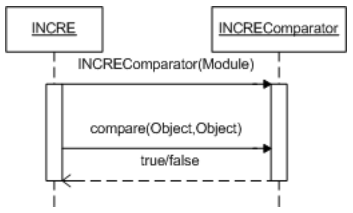


Figure 6.5: UML sequence diagram of comparing objects

The interaction starts when `INCRE` creates a comparator for a specific compilation module. Then `INCRE` asks the comparator to compare two objects. The comparator compares the two objects and returns false in case the objects differ and true if not. Figure 6.6 shows an UML activity diagram of the comparator.

The first task of the comparator is counting the number of comparisons. Therefore, it increases its internal counter by one. The total number of comparisons is a good indication for the effec-

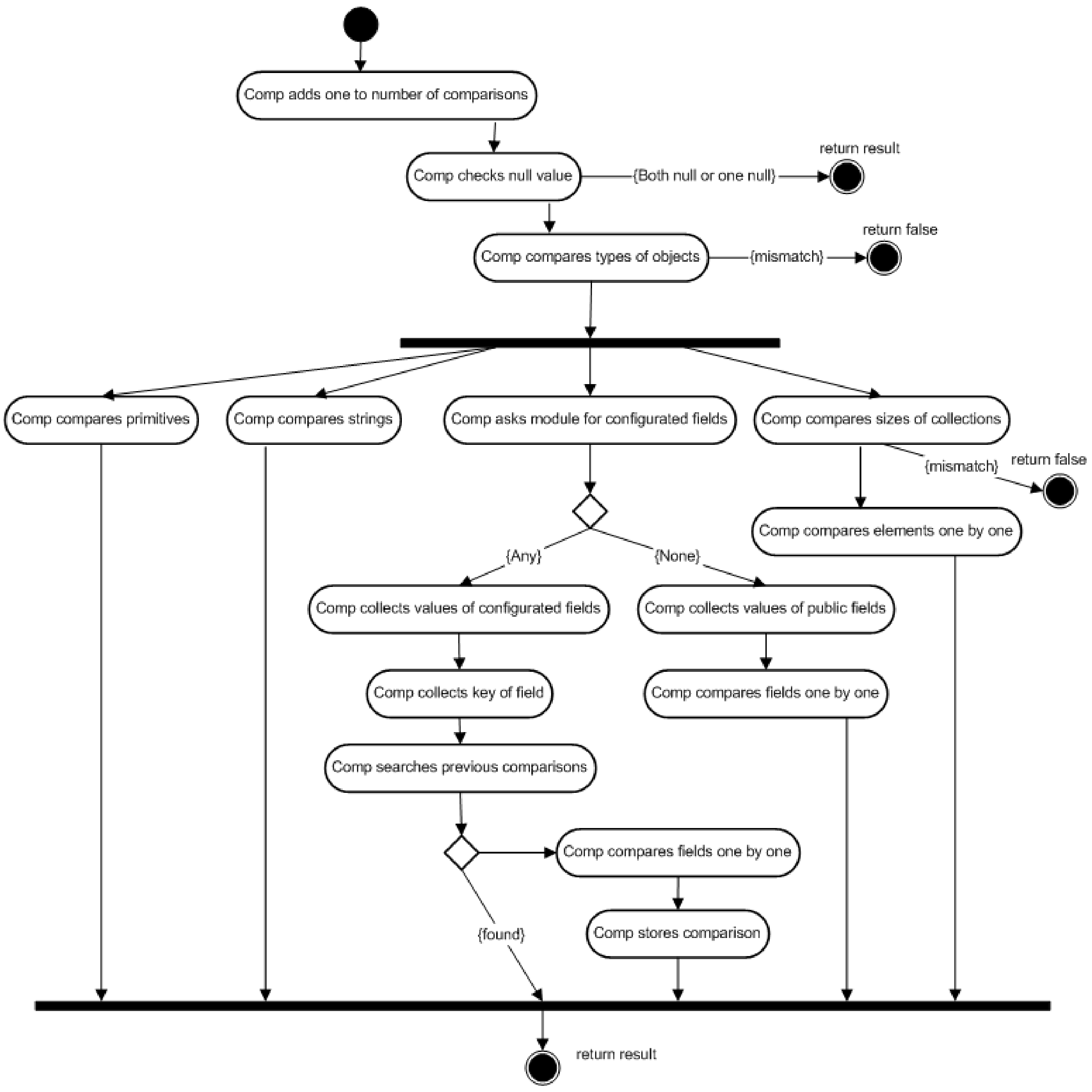

Comp adds one to number of comparisons
Comp checks null value
{Both null or one null}
return result
Comp compares types of objects
{mismatch}
return false
Comp compares primitives
Comp compares strings
Comp asks module for configurated fields
Comp compares sizes of collections
{mismatch}
return false
Comp compares elements one by one
{Any}
{None}
Comp collects values of configurated fields
Comp collects values of public fields
Comp collects key of field
Comp compares fields one by one
Comp searches previous comparisons
Comp compares fields one by one
{found}
Comp stores comparison
return result

Figure 6.6: UML activity diagram of comparing objects

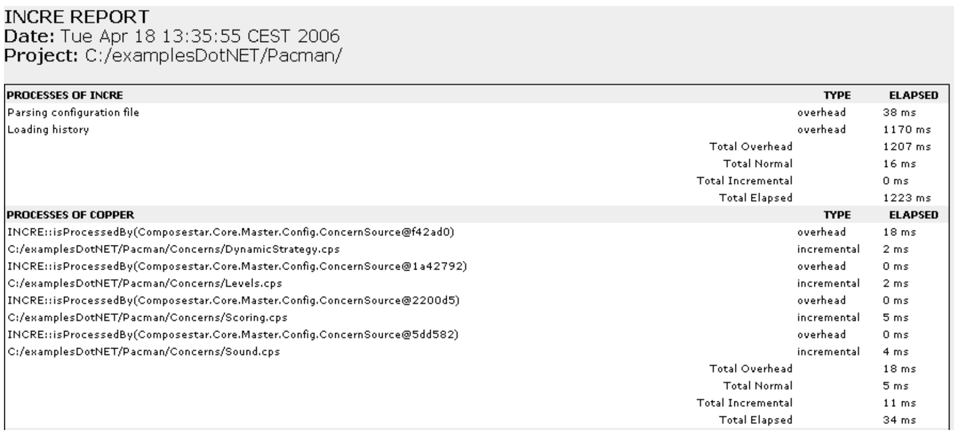
tiveness of the comparator. Keeping the number of comparisons low also means low overhead costs. After increasing the counter, the comparator follows its algorithm of comparing two objects. The steps of this algorithm are:

1. Comparing the objects on `null` value. The comparator returns `false` in case one object is null and `true` in case both are null. The comparator continues with step two in case both objects are not null.
2. Comparing the type of objects. When there is a mismatch in types, the comparator returns `false`. Otherwise is continues with step three.
3. This step depends on the type of the objects that are compared. The comparator recognizes four different cases:
   (a) Two primitives are compared with standard Java `"=="` operator.
   (b) Two strings are compared with standard Java `equals()` method.
   (c) For collections, the comparator first compares both sizes. If there is a mismatch in size, then it returns `false`. Otherwise, it iterates over the collection and compares the elements one by one. Two elements are compared by making a recursive call to the comparator. If two elements differ from each other, then the comparator returns `false`.
   (d) For all other types the comparator continues with step four.
4. Collecting the fields of the type to compare. In the XML configuration file, each compilation module may have been configured with a list of comparisons. Each comparison consists of a type and a list of fields. The comparator asks the compilation module extracted from the XML configuration file for his list of comparisons. When this list contains an entry for the type, the comparator collects the values of the type's configured fields by using reflection. In case the XML does not contain an entry, the comparator collects all public fields of the type and its parent(s).
5. The final step is comparing the collected fields. For fields not collected from the XML file, the comparator simply compares the values one by one by making a recursive call to the `compare` method. When it finds a mismatch, the comparator stops comparing and returns `false`. In case all fields are equal, the comparator returns `true`. For fields extracted from the XML file, the comparator uses an optimization. Each comparison has a unique key namely the hashcodes of both fields combined. Before comparing the fields, the comparator checks its list of comparison results to see whether it has already completed the comparison before. If this is the case, then the comparator uses this result instead of comparing again. Otherwise, the comparator compares the fields. After comparing the fields, the comparator adds the result of the comparison to its internal list. This way the comparator minimizes the number of comparisons.

6.4 Reporting

We have implemented a reporter that generates a timing report in HTML format. Figure 6.7 shows a snapshot of an example report. The report contains timings of processes performed by a compilation module. A process consists of a description, timing in milliseconds and a category. There are three timing categories:

- Normal: for timing a non-incremental process.
- Incremental: for timing an incremental process.
- Overhead: for timing a process that causes overhead.



INCRE REPORT
Date: Tue Apr 18 13:35:55 CEST 2006
Project: C:/examplesDotNET/Pacman/

PROCESSES OF INCRE	**TYPE**	**ELAPSED**
Parsing configuration file	overhead	38 ms
Loading history	overhead	1170 ms
Total Overhead		1207 ms
Total Normal		16 ms
Total Incremental		0 ms
Total Elapsed		1223 ms
PROCESSES OF COPPER	**TYPE**	**ELAPSED**
INCRE::isProcessedBy(Composestar.Core.Master.Config.ConcernSource@f42ad0)	overhead	18 ms
C:/examplesDotNET/Pacman/Concerns/DynamicStrategy.cps	incremental	2 ms
INCRE::isProcessedBy(Composestar.Core.Master.Config.ConcernSource@1a42792)	overhead	0 ms
C:/examplesDotNET/Pacman/Concerns/Levels.cps	incremental	2 ms
INCRE::isProcessedBy(Composestar.Core.Master.Config.ConcernSource@2200d5)	overhead	0 ms
C:/examplesDotNET/Pacman/Concerns/Scoring.cps	incremental	5 ms
INCRE::isProcessedBy(Composestar.Core.Master.Config.ConcernSource@5dd582)	overhead	0 ms
C:/examplesDotNET/Pacman/Concerns/Sound.cps	incremental	4 ms
Total Overhead		18 ms
Total Normal		5 ms
Total Incremental		11 ms
Total Elapsed		34 ms

Figure 6.7: Example HTML timing report

Listing 6.1 shows a code snippet for timing any process. First, we acquire an instance of `INCREReporter` (lines 1 – 2). Then, we start the timing of the process by calling the reporter's `openProcess` method. This method requires three parameters: the name of the compilation module that performs the process, the description of the process and the type of process. By receiving this call, the reporter creates and starts a timer. At the end of the process, we stop the timer (line 8) and the reporter subsequently stores the elapsed time. Finally, all timed processes are printed in HTML format as soon as the reporter is closed (line 9).

```
1  INCRE incre = INCRE.instance(); // get an instance of INCRE
2  INCREReporter reporter = incre.getReporter(); // get a reporter
3
4  //start timing process
5  INCRETimer timer = reporter.openProcess(
6              "MODULE","PROCESS_DESC",INCRETimer.TYPE_OVERHEAD);
7  ... // continue process
8  timer.stop(); // stop timing
9  reporter.close(); // close reporter
```

Listing 6.1: Code snippet for timing a process

To provide a means to analyze the difference between the non-incremental and incremental compilation, we included the following processes in the HTML report. Firstly, for each compilation module, we timed each processing of an input object. Secondly, to find the bottlenecks of a compilation module, we timed some individual parts of a compilation module. Finally, we timed INCRE's `isProcessedByModule` method to find out how much compilation time (overhead) INCRE consumes to fulfill incremental compilation.

6.5 Control Flow

This section describes the control flow of incremental compilation. Figure 6.8 shows an UML sequence diagram of the control flow. The diagram defines the following sequence of actions:

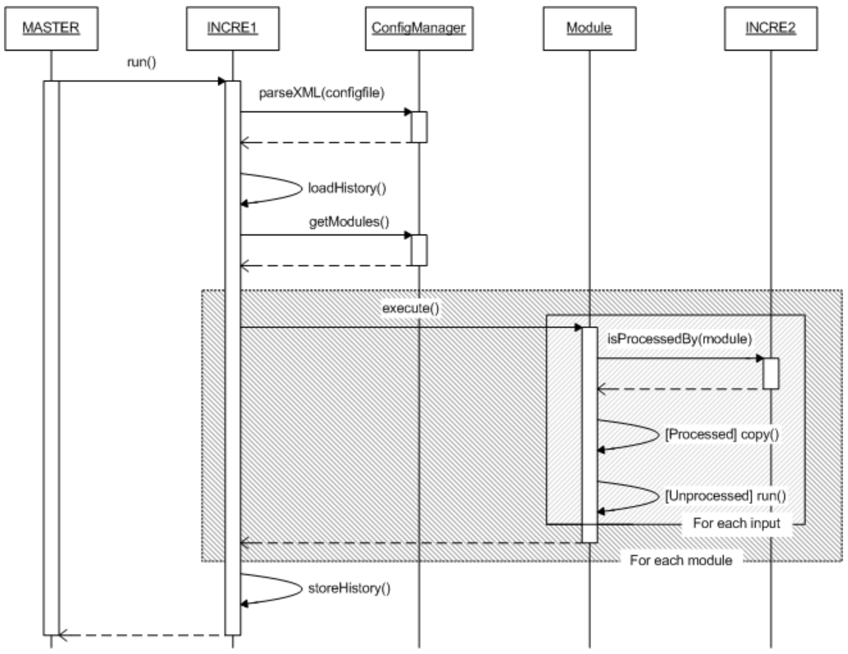


Figure 6.8: UML sequence diagram - control flow of incremental compilation

1. **Start `INCRE` module.** The `MASTER` calls the `run()` method to start `INCRE`[1].

2. **Collect configurations by parsing the XML configuration file.** An instance of `ConfigManager` parses the XML configuration file called *"INCREConfig.xml"*. (see Section 6.2).

3. **Load the historical compilation results.** An instance of `INCRE` is responsible for retrieving the repository of a previous compilation of the same project. To accomplish this, it serializes the repository to a file called *"history.dat"*. `INCRE` uses standard Java deserialization to reconstruct the 'history' repository.

4. **Run the Compose⋆ compilation modules.** An instance of `INCRE` first asks its `ConfigManager` for all compilation modules extracted from the XML file. Then, for each compilation module its `run()` method is called.

5. **Run normal or copy operation of a compilation module.** Each compilation module processes a set of input objects. For each input object, a compilation module can use `INCRE's` service called `isProcessedBy`. This service tells the compilation module whether it has already processed an input object in an earlier compilation or not. If so, the compilation module does not need to reprocess the input object. Instead, it can search the backup repository for the result of that previous process. The `copy` operation of the

[1]Note that this sequence diagram contains two instances of `INCRE`. This is only done for visibility purposes. In reality, there is only one instance of `INCRE`.

compilation module realizes this. The service `isProcessedBy` is further explained in Section 6.5.1.

6. **Store the compilation results.** The incremental compilation ends with serialization of the repository by invoking the `storeHistory` method of `INCRE`. This method uses standard Java serialization to dump the repository into a *"history.dat"* file.

6.5.1 IsProcessedBy service

This section describes the control flow of the service `isProcessedBy`. Figure 6.9 shows an UML sequence diagram of the control flow. The diagram defines the following sequence of actions:

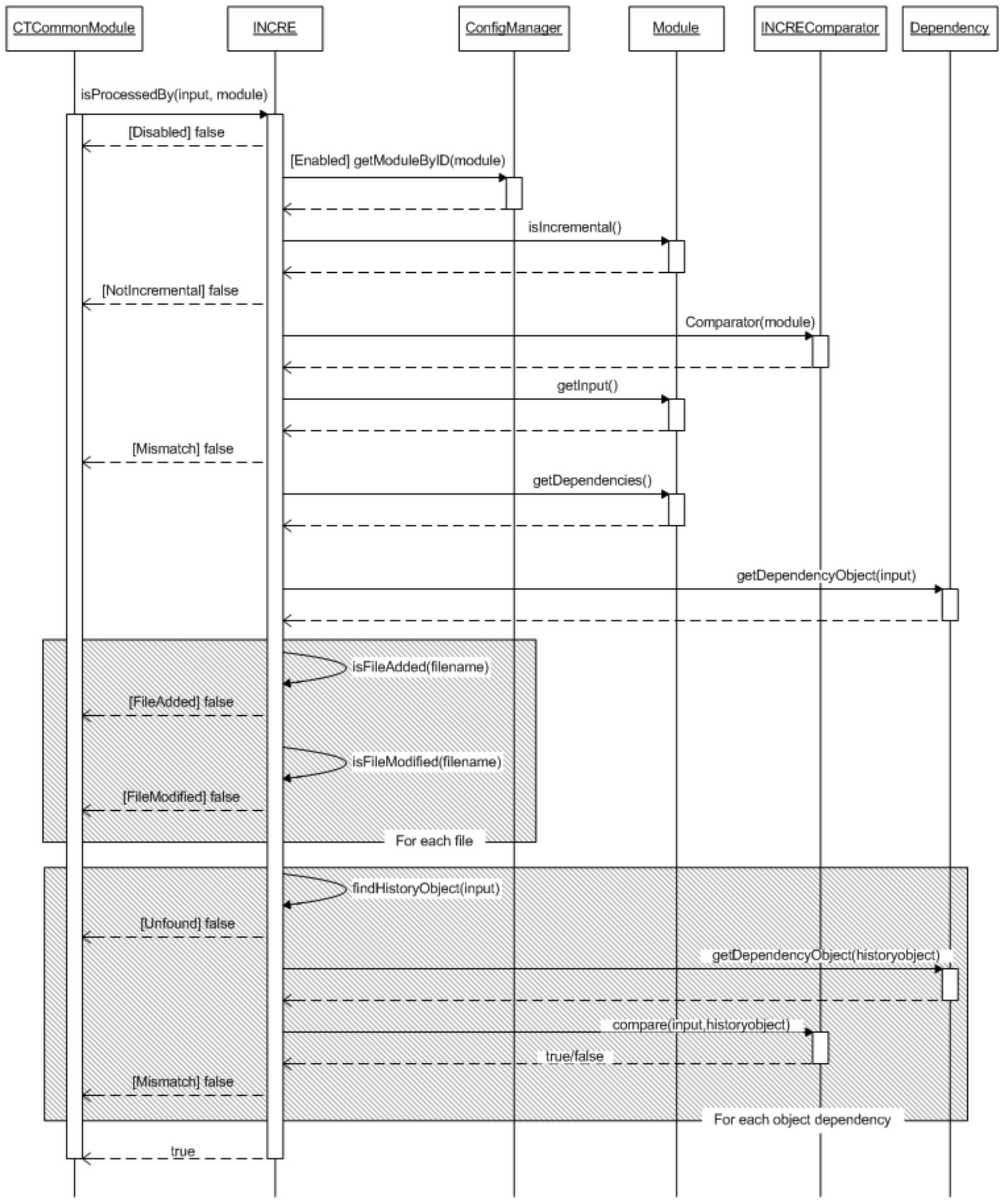


Figure 6.9: UML sequence diagram - control flow of operation `isProcessedBy`

1. **Start service.** The service starts when `INCRE` receives a call from a compilation module. This call has two parameters, the input object to be checked and the name of the calling compilation module.

2. **Verify if calling module is incremental.** There is a project configuration for enabling or disabling incremental compilation. If this configuration is off, then `INCRE` does not continue and returns `false` (input unprocessed). In addition, each compilation module has an incremental compilation option. `INCRE` asks `ConfigManager` for the compilation module and retrieves the value of this option. If the option is off, then `INCRE` stops and returns `false`.

3. **Initialize comparator.** Before comparing objects for a compilation module, the comparator needs to know for which compilation module it is comparing. Therefore, we call the constructor of class `INCREComparator`. This constructor has one parameter, the name of the compilation module to compare for.

4. **Verify input.** `INCRE` is enhanced with input verification. The type of the input object is compared with the configured value of the `Module`. If there is a mismatch, then `INCRE` logs an error message and returns `false`. Otherwise, it continues with the next step.

5. **Check data dependencies.** `INCRE` checks the data dependencies in the following steps:

   (a) **Retrieve the configured dependencies of a module.** Each compilation module has a set of `dependency` objects obtained after parsing the XML configuration file. `INCRE` asks the calling compilation module for this set.

   (b) **For each dependency object, collect the value used in the current compilation cycle.** `INCRE` achieves this by calling the `getDependencyObject` method. This method has one parameter: the input object of the calling compilation module. Figure 6.10 shows the control flow of the `getDependencyObject` method. It shows that each dependency has a path of nodes. The value of a dependency is obtained by following this path. Each node in the path expects one input object and returns one output object. The output of one node is passed on to the next node. The first node receives the input of the calling module. The object returned by the last node is the value of the dependency.

   (c) **For file dependencies, perform timestamp-comparison.** In case the value of a dependency is a file, `INCRE` firstly checks whether the file is available in the stored project configurations (`isFileAdded`). If not available, then the file is newly added and thus marked as unprocessed. If available, then `INCRE` checks the timestamp of the file (`isFileModified`). If the file is older than the completion timestamp of the previous compilation, then `INCRE` marks the input as processed. Otherwise, the file is possibly modified. Therefore, the input is marked as unprocessed.

   (d) **For object dependencies, perform object-comparison.** `INCRE` performs object comparison in the following steps:

      i. **Retrieve the stored input object**. Each input processed by a compilation module has a unique identifier. `INCRE` searches the history repository for an object with the same identifier (`findHistoryObject`). If it does not find such an object, then `INCRE` marks the input as unprocessed. Otherwise, it continues with the next step.

ii. **Collect the value of the dependency used in a previous compilation cycle.** `INCRE` calls `getDependencyObject` of the checked dependency for the found history object. In addition, all nodes in the path use the stored repository instead of the newly built one.

iii. **Compare the two collected dependency objects.** Therefore, `INCRE` calls the `compare` method of `INCREComparator`. Section 6.3 elaborates on the comparing mechanism. In case the comparator finds a mismatch, `INCRE` stops and marks the input of the compilation module as unprocessed. Otherwise, `INCRE` checks the next dependency.

6. **Finish service.** `INCRE` returns true when it has checked all data dependencies and did not find any mismatches. The compilation module now knows that it is safe to execute the copy operation instead of its slower regular operation.

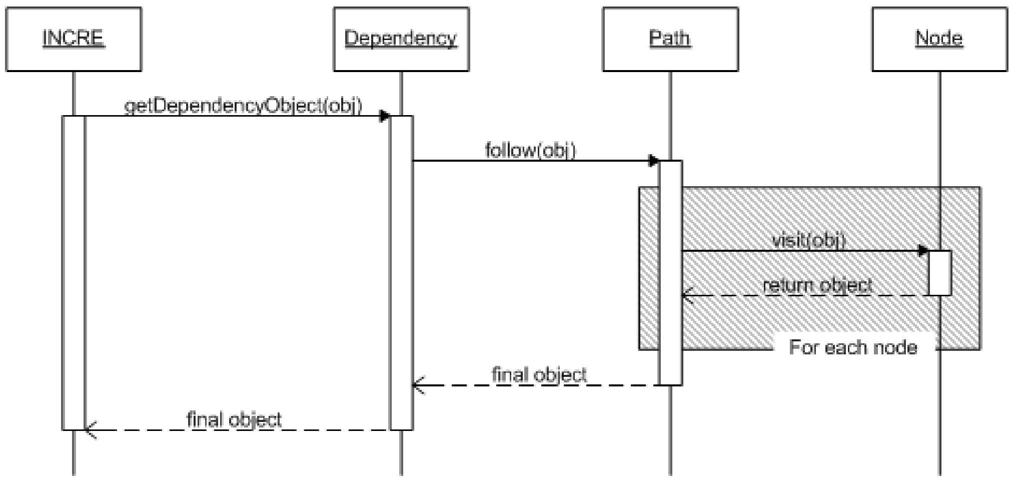


Figure 6.10: UML sequence diagram - control flow of retrieving a data dependency

6.6 Conclusion

This chapter presented the implementation of package `INCRE`, which introduces incremental compilation to Compose⋆. The next chapter elaborates on how this implemented service is configured and used by several Compose⋆ compilation modules.

I had been told that the training procedure with cats was difficult. It is not. Mine had me trained in two days.
– Bill Dana

Chapter 7

Realization of Incremental Compose⋆ Compilation Modules

Chapter 5 concluded with a five-step procedure for enhancing Compose⋆ compilation modules with incremental performance. In this chapter and Appendix C, we perform this procedure for eight chosen compilation modules.

7.1 Compilation Modules Enhanced

Section 3.2 describes that the non-incremental Compose⋆/.Net compilation process contains sixteen compilation modules. From these sixteen compilation modules, we have chosen to enhance the following eight ones with incremental performance:

COPPER - Compose⋆ Parser	FILTH - Filter Composition & Checking
HARVESTER - Type Harvesting	SECRET - Semantic Reasoning Tool
COLLECTOR - Type Collector	RECOMA - Source Compiler
LOLA - Logic Language	ILICIT - Interception Inserter

We have chosen to enhance these compilation modules, because they generally consume the most compilation time and resources. Thus, adding incremental performance to these compilation modules likely provides the best chance of decreasing compilation time.

Throughout the remainder of this chapter, we use the compilation modules FILTH and RECOMA as examples to clarify the five-step procedure. For a detailed description of the other six compilation modules, we refer to Appendix C.

7.2 Step One: Identifying Input, Output and Data Dependencies

In our incremental rebuilding solution, exact knowledge of a compilation module's input, output and data dependencies is indispensable to minimize the redundant operations of a compilation module. Hence, the first step for enhancing a compilation module with incremental performance is identification of these three characteristics. To identify these three characteristics, we manually analyzed the compilation modules and filled in the following scheme.

Input: - What are the input objects of the compilation module?
Processing: - How processes the compilation module its input to output?
Output: - What is the output of the compilation module?
Dependencies: - Which data (e.g., objects and files) influence the way the compilation module processes a certain input object?
Motivation Incremental Performance - Why should we enhance the compilation module with incremental performance?

The next two subsections present the results of our manual analysis of the compilation modules FILTH and RECOMA. We refer to Appendix C for the analyses of the remaining six compilation modules.

7.2.1 FILTH Analysis

The results of our analysis of FILTH are:

Input: Repository with superimposition resolved and a filter ordering specification file that contains orderings constraints in XML format.

Processing: FILTH iterates over all concerns with one or more filtermodules superimposed on. For each concern, it calculates all possible filtermodule orderings. A filtermodule ordering is an ordered list containing the names of superimposed filtermodules. The user can put constraints on the orderings by configuring an XML ordering specification file.

Output: Two objects attached to the input concern. The first is called "SingleOrder" and contains a first ordering of filtermodules superimposed on the concern. The second one is called "FilterModuleOrders" and contains all possible orderings of filtermodules superimposed on the concern.

Dependencies: The following two types of data influence FILTH's processing of an input concern:

1. The names of all filtermodules superimposed on the input concern. It is trivial that a change to a set consequently affects the possible orderings of that set.
2. The parts of the filter ordering specification file that relate to the input concern. When the user makes structural changes to these parts, he can introduce or remove constraints on the filtermodule orderings for the input concern. This may affect the set of possible filtermodule orderings for the input concern.

Motivation Incremental Performance FILTH recalculates all possible filtermodule orderings in every compilation. This strategy is inefficient as it leads to redundant repeats of calculations. By applying our incremental rebuilding solution to FILTH, we should minimize these redundant recalculations and improve FILTH's performance.

7.2.2 RECOMA Analysis

The results of our analysis of RECOMA are:

Input: User-provided .NET sources and compiled dummy sources with full signatures (produced by compilation module ASTRA).

Processing: RECOMA iterates over all user-provided .NET sources and compiles them against the dummy assemblies with full signatures [27]. To compile the sources, RECOMA uses the standard Microsoft .NET compilers [3].

Output: Compiled .NET sources stored on disk.

Dependencies: The following five data influence RECOMA's compilation of a source:

1. Compiled source produced in the previous compilation. If the compiled version of the input source is missing, due to deletion or replacement, then RECOMA needs to recompile the input source.
2. Structural content of the input source. Structural changes to the input source may introduce compilation errors and unavoidably affects the compilation results. Examples of structural changes are: addition of methods or renaming fields. Examples of not structural or meaningless changes are changes to comments, tabs and spaces.
3. Structural content of referenced sources. A source may depend on the content of another source. An example of this is the signature of an interface. People often declare an interface in another source than its implementation. A change to the signature of an interface may introduce compilation errors when compiling the source containing its implementation.
4. Structural content of referenced libraries. A source may, similar to referenced sources, depend on the content of referenced libraries.
5. Full signatures of concerns extracted from referenced sources. A source is compiled by calling the correct .NET compiler and using the dummy assembly (compiled dummy sources) as link input. The reason for this is to overcome the signature mismatch problem described in [27]. The dummy assembly contains the full signatures of concerns declared in all user-provided sources. However, not all of these full signatures influence the compilation of a source. Note that the full signatures of concerns extracted from the input source itself are not a dependency. This is because, while compiling, a .NET compiler uses the original signature of the input source instead [27]. In addition, full signatures of concerns extracted from unreferenced sources are not a dependency because they are simply not referenced by the input source and thus not used when compiling the input source.

Motivation Incremental Performance RECOMA recompiles all sources in every compilation. This strategy is inefficient as it leads to redundant repeats of compilations. By applying our incremental rebuilding solution to RECOMA, we should minimize these redundant recompilations and improve RECOMA's performance.

7.3 Step Two: Serializing Repository

Our incremental rebuilding solution requires that the input, output and dependent data of a compilation module are kept between compilations (see Section 4.3). To realize this, INCRE provides two methods called `storeHistory` and `loadHistory`. These methods use Java's standard object serialization to read and write the whole Compose⋆ repository to disk. However, to be able to use these methods, we must meet the following requirement.

Each object to be serialized must implement the `java.io.Serializable` interface or inherit that implementation from its object hierarchy.

In this second step, we verify whether the above requirement is met for all input, output and dependent data of a compilation module. This seems tedious but it is not because of the following reasons. Firstly, most objects stored in the repository inherit from the superclass `RepositoryEntity`. Secondly, we implemented the `java.io.Serializable` interface for this superclass. By doing this, we fulfilled the above requirement for all objects that inherit from this superclass. This makes this step easier as most objects stored in the repository already fulfill the above requirement.

7.4 Step Three: Implementing Copy Operation

Step three of our incremental rebuilding solution is the implementation of the so-called copy operation. This copy operation should copy the output belonging to a specific input object from the backup repository into the repository being built. In order to safe compilation time, this copy operation should be faster than an original processing of an input object. In the next two subsections, we present the implementation of the copy operations for the compilation modules FILTH and RECOMA.

7.4.1 Copy Operation FILTH

In Section 7.2.1, we concluded that the input of FILTH are concerns and that the output of FILTH are two calculated objects attached to these concerns. Hence, the copy operation of FILTH should do the following. Search the backup repository for the two calculated objects belonging to a specific concern and copy them into the repository being built. Listing 7.1 presents a copy operation that achieves this.

```java
public void copyOperation(Concern c)
{
  INCRE inc = INCRE.instance();

  // Copy dynamic objects 'FilterModuleOrders' and 'SingleOrder'
  Concern cop = (Concern)inc.findHistoryObject(c);

  LinkedList forders = (LinkedList)cop.getDynObject("FilterModuleOrders");
  c.addDynObject("FilterModuleOrders",forders);

  FilterModuleOrder fmorder = new FilterModuleOrder((LinkedList)forders.getFirst());
  c.addDynObject("SingleOrder",fmorder);
}
```

Listing 7.1: Copy Operation FILTH

This copy operation has one parameter: the input concern of FILTH which should be copied for. The method block of FILTH's copy operation contains six statements. The first statement obtains the singleton instance of class `INCRE` by calling the `instance()` method of class `INCRE` (line 3). After obtaining an instance of `INCRE`, the copy operation asks it to search the backup repository for a copy of the input concern. It obtains a copy of the input concern by calling INCRE's `findHistoryObject` method (line 6). Finally, the copy operation retrieves the two calculated objects from this copy and attaches them to the input concern by calling its `addDynObject` method (lines 8–12).

7.4.2 Copy Operation RECOMA

For RECOMA, we do not need to implement a copy operation. The reason for this is the following. Recall that the copy operation only applies to compilation modules that produce one or more repository objects (see Section 5.5). In Section 7.2.2, we concluded that the output of RECOMA only consists of compiled sources stored on disk. This means that RECOMA does not produce any repository objects. Hence, we do not need to implement a copy operation for RECOMA.

7.5 Step Four: Configuring Data Dependencies

Step four of our incremental rebuilding solution is the configuration of the data dependencies of a compilation module. By configuring these data dependencies, we specify which data INCRE should compare to answer the main question of our incremental rebuilding solution: *Does an input object need to be processed again by the compilation module?* For a detailed description of how INCRE uses these configurations to answer this main question, we refer to Section 6.5.

To configure the data dependencies of a compilation module, we can use our XML configuration file. This XML configuration file supports several XML tags for configuring the data dependencies. For a detailed description of these XML tags, we refer to Appendix A. In the next two subsections, we present our dependency configurations for the compilation modules FILTH and RECOMA.

7.5.1 Dependency Configuration FILTH

From FILTH's analysis (see Section 7.2.1), we conclude that there are two data types that influence the possible filtermodule orderings for a concern. These are the user-provided configuration file, which puts constraints on the filtermodule orderings, and the names of all superimposed filtermodules. Listing 7.2 presents a configuration of these two data dependencies.

```xml
<dependencies>
  <dependency type="FILE" name="specfile">
    <path>
      <node type="CONFIG" nodevalue="FILTH_INPUT"></node>
    </path>
  </dependency>
  <dependency type="OBJECT" name="fmodules">
    <path>
      <node type="DYNAMIC" nodevalue="superImpInfo"></node>
      <node type="FIELD" nodevalue="theFmSIinfo"></node>
```

```
11      </path>
12    </dependency>
13  </dependencies>
```

Listing 7.2: Dependency Configuration of FILTH

The first configured dependency is the user-provided configuration file. This file is referenced by a project configuration called ”FILTH_INPUT”. To retrieve this file, we only need to specify a dependency of type ”FILE” containing a path of one config node (lines 2-6).

The second configured dependency refers to all superimposed filtermodules on a concern. These superimposed filtermodules are referenced by the field ”theFmSIinfo” of a concern’s dynamic object called ”superImpInfo”. To retrieve these objects, we only need to specify one dependency of type ”OBJECT” containing a path of one dynamic node and one field node (lines 8-11). Note that this second dependency returns whole filtermodules instead of only their names. In Section 7.6.1, we explain how this can be further optimized.

7.5.2 Dependency Configuration RECOMA

From RECOMA’s analysis (see Section 7.2.2), we conclude that there are five dependencies that influence RECOMA’s compilation of a user-provided source. We have configured four of these dependencies in our XML configuration file. These four dependencies are the content of the input source, the content of all its referenced sources, the content of referenced assemblies and the full signatures of concerns extracted from a referenced source. Listing 7.3 presents a configuration of these four dependencies.

```
1   <dependencies>
2     <dependency type=”FILE” name=”source”>
3       <path>
4         <node type=”FIELD” nodevalue=”fileName”></node>
5       </path>
6     </dependency>
7     <dependency type=”FILE” name=”xternals”>
8       <path>
9         <node type=”METHOD” nodevalue=”Composestar.DotNET.COMP.DotNETCompiler.
              externalSources”></node>
10      </path>
11    </dependency>
12    <dependency type=”FILE” name=”CompileLibsDependencies”>
13      <path><node type=”CONFIG” nodevalue=”Dependencies”></node></path>
14    </dependency>
15    <dependency type=”OBJECT” name=”fullsignatures”>
16      <path>
17        <node type=”METHOD” nodevalue=”Composestar.DotNET.COMP.DotNETCompiler.
              fullSignatures”></node>
18      </path>
19    </dependency>
20  </dependencies>
```

Listing 7.3: Dependency Configuration of RECOMA

The first dependency returns the filename of an input source. This filename is referenced by a field ”fileName” of the input source (line 4). The second dependency returns the filenames of sources referenced by an input source. These filenames are obtained by a new helper method called ”externalSources” (line 9). The third dependency returns the filenames of all assemblies referenced by a user-provided base program. These filenames are referenced by a project configuration called "Dependencies" (line 15). The last dependency returns the full signatures of all concerns extracted from sources referenced by an input source. These signatures are obtained by a new helper method called "fullSignatures" (line 17).

7.6 Step Five: Optimizing Repository Entity Comparison

As described in Section 5.3.2, our incremental rebuilding solution requires a comparator for comparing repository objects. In Section 6.3, we described an implementation of this comparator called `INCREComparator`. Below, we briefly summarize the working procedure of this comparator.

When a compilation module asks the comparator to compare two objects, it first retrieves and compares the types of these objects. If there is a mismatch in these types, it returns `false`. Otherwise, it continues with the next step. In this second step, the comparator verifies whether the user has configured so-called *field-restrictions* for the type and calling compilation module. If this is the case, then the comparator will compare the two objects only on these configured fields. Otherwise, the comparator will compare the two objects on all their fields by default.

So, in other words, by configuring field-restrictions, the user can control and optimize the comparing mechanism. This optimization of the comparator is the final (optional) step of our rebuilding solution. To clarify this optimization step further, the next two subsections present our comparator optimizations for FILTH and RECOMA. We refer to Appendix C for comparator optimizations of the remaining six compilation modules.

7.6.1 Comparison Configuration FILTH

FILTH's second dependency returns a list of filtermodules rather than only the names of the filtermodules. This means that it returns more data than we actually need. This slows down our comparing mechanism, because the comparator will compare whole filtermodules instead of only their names. Therefore, we want to restrict the comparison of filtermodules further. We achieve this by configuring a field-restriction for filtermodules. Listing 7.4 shows this restriction.

```
<comparisons>
  <type fullname="Composestar.Core.CpsProgramRepository.CpsConcern.References.
      FilterModuleReference">
    <field name="name"></field>
  </type>
</comparisons>
```

Listing 7.4: Comparison Configuration of FILTH

The above configuration restricts the comparison of instances of `FilterModuleReference` to their field called "name". This means that when we compare two instances of `FilterModuleReference`, we only compare on their names and ignore their other redundant data.

7.6.2 Comparison Configuration RECOMA

RECOMA's fourth configured dependency returns a list of signatures. However, not all fields of these signatures might influence RECOMA's compilation of a source file. This means that it returns more data than we actually need. Again, we can restrict this data further by means of our comparison configuration. Listing 7.5 presents a configuration of this restriction. This configuration restricts the comparison of signatures to only the fields that may influence RECOMA's compilation of a source file.

```
comparisons>
  <type fullname="Composestar.Core.CpsProgramRepository.Signature">
    <field name="methodByName"></field>
  </type>
  <type fullname="Composestar.Core.CpsProgramRepository.MethodWrapper">
    <field name="RelationType"></field>
  </type>
</comparisons>
```

Listing 7.5: Comparison Configuration of RECOMA

7.7 Conclusion

This chapter and Appendix C described in detail how eight Compose⋆ compilation modules have been enhanced with incremental strategies. In the next chapter, we evaluate the efficiency of these implemented incremental strategies by charts of tests.

Chapter 8

Evaluation

In this chapter, we evaluate the efficiency of the strategies described in the previous chapter by means of charts. The organization of this chapter is as follows. The charts presented in this chapter are the results of numerous timing tests. To clarify on this, we first describe which and how we have executed these timing tests. After this clarification, we give an overview of the charts presented in this chapter and summarize the expected results. Finally, we present the charts and explain any unexpected results.

8.1 Test Cases and Conditions

This section defines test cases and test conditions used for testing the performance of the incremental compiler.

8.1.1 Cases

To test the performance of the incremental compiler, we have used the following tests cases:

Four different example projects. To test the scalability of the compiler, we have used four different example projects scaling from very small to large. Table 8.1 shows the characteristics of these examples.

Table 8.1: Characteristics of test examples

Name	Size	Source files	Lines of Code	Classes	FilterModuleOrders
EX1	Very Small	4	199	4	1
EX2	Small	21	2170	21	11
EX3	Medium	107	2872	107	124
EX4	Large	344	68459	472	731

Six different scenarios. To test the performance of the incremental compiler, we have chosen for the following six scenarios.

S1 Compilation after no modifications to base program and aspects.

S2 Compilation after modifying the signature of one concern (e.g., adding a method).
S3 Compilation after adding a logging concern.
S4 Compilation after modifying one method block (e.g., adding a print statement).
S5 Compilation after adding one new joinpoint.
S6 Compilation after modifying one predicate selector into an equivalent one.

Note that there are numerous more scenarios to think of. However, our intention is not to be complete but rather give an indication for a few distinctive and common scenarios.

Non-incremental versus incremental. To recognize the differences between non-incremental and incremental compilation, we have executed the above combinations of examples and scenarios for both the non-incremental and incremental compiler.

To increase the reliability of the tests, we have conducted each test case ten times and averaged the results. This brings the total number of tests to 480 [1].

8.1.2 Conditions

In order to have the same conditions for all tests, we have executed all tests on one system. Table 8.2 shows the configuration of this system.

Table 8.2: Configuration of the system used for testing

Central Processing Unit	AMD Athlon(tm) 64 Processor 3000+ 2.00 GHz
Memory	512 MB of RAM
Operating System	Windows XP
Java VM Options	-Xmx512m -Xms512m -Xmn128m

8.2 Charts and Expected Results

In this chapter, we present the following charts:

1. **Share of compilation modules in non-incremental compilation time.** This chart shows, for each example, which compilation modules are relatively fast or slow during non-incremental compilation of that example. This information helps us in deciding for which compilation modules to introduce or maintain incremental performance. The reason for this is the following. The more compilation time a compilation module consumes, the bigger the chance we can decrease this consumption of compilation time by means of incremental performance. In this chart, we also expect to see differences in scalability between compilation modules because of differences in complexity and responsibility. This information tells us which compilation modules need improvement to ensure scalability of Compose⋆ compilation.

[1] four (examples) multiplied by six (scenarios), multiplied by two (incremental and non-incremental) and finally multiplied by ten (average results)

2. **Performance improvement by example and scenario.** This chart shows, for each combination of example and scenario, the performance improvement of the incremental compiler compared to the non-incremental compiler. This information tells us in which cases it is beneficial to use the incremental compiler and which not. In this chart, we expect to see the following two results:

   (a) Out of all scenarios, we should realize the biggest performance improvement in the first scenario. We expect this because of the following. INCRE only recognizes an operation as redundant when none of its dependent data has been modified. In the first scenario nothing has been modified, thus INCRE should find the most redundant operations in this scenario. For each operation recognized as redundant, INCRE calls the copy operation of a compilation module instead of its regular operation. These copy operations are intended to be faster than the regular operations. Thus, the more redundant operations INCRE finds, the faster the compilation will be. Because the first scenario should have the most redundant operations, this scenario should also have the fastest compilation. This explains why we should realize the biggest performance improvement in the first scenario.

   (b) We expect to see a positive performance improvement in all test cases. We expect this because of the following. In all test cases, we only perform limited (zero or one) changes to the examples rather than many changes. Because of this limited changes, we expect that the incremental compiler should be able to find enough redundant operations to speed up compilation of the examples.

3. **Overhead by example and scenario.** This chart shows the *overhead* created by the incremental compiler for each combination of example and scenario. By *overhead*, we mean compilation time consumed by the compilation module INCRE to fulfill its service, incremental compilation. This information gives us insight into the maximum and minimum performance improvement of the incremental compiler compared to the non-incremental compiler. To be precise, the maximum performance improvement of the incremental compiler is hundred percent minus the maximum overhead of INCRE. In addition, the minimum performance improvement is zero percent minus the maximum overhead of INCRE. In this chart, we expect to see the following result:

   (a) Out of all scenarios, we should realize the maximum overhead in the first scenario. We expect this because of the following. INCRE's overhead consists of two tasks. The first task is serialization and deserialization of the repository. The second task is dependency checking (acquire two sets of dependent data and compare them). To complete the latter task, INCRE uses the following strategy. It starts by checking the first dependent data of a compilation module. When this dependent data has not been modified since the previous compilation of the same project, INCRE moves on to next dependent data. Otherwise, it stops dependency checking. In other words, the more changes a programmer makes to its program, the less dependency checks are performed by INCRE. In the first scenario, nothing has been modified. This is in contrast to the other scenarios, in which we do perform changes. This explains why we should realize the biggest overhead in the first scenario.

4. **Average performance improvement and overhead of compilation modules.** This chart shows the average (of each example and scenario) performance improvement and overhead of the eight incremental compilation modules and INCRE. This information tells us which compilation modules profit the most from incremental compilation. In this chart, we expect to see differences between compilation modules because of differences in complexity and responsibility.

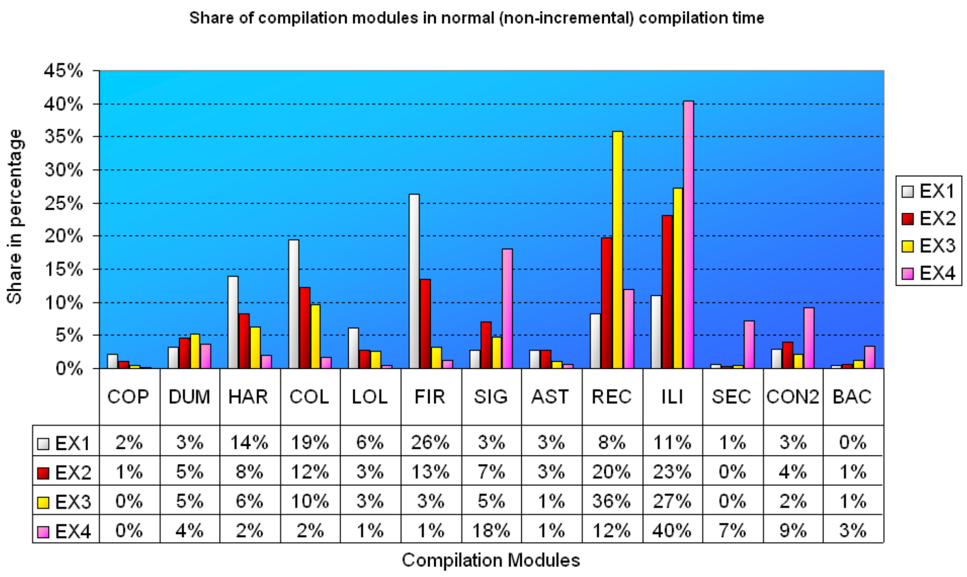
5. **Performance improvement and overhead of a compilation module by example and scenario.** This chart shows, for each combination of example and scenario, the performance improvement and overhead of a compilation module. This information tells us whether for a specific case, it is beneficial to turn on the incremental performance of a compilation module or not. We present this type of chart for the compilation module INCRE and the eight incremental compilation modules. We expect to see performance improvement for incremental compilation modules in almost all cases. Exceptions to this are the following cases:

   (a) **FILTH and the third scenario.** In the third scenario, we add a logging concern to the examples. This logging concern implies creating a new filtermodule that is superimposed on each concern of the user-provided base program. For FILTH, this has the following consequence. In Section 7.2.1, we described that one of the data dependencies of FILTH are the names of all superimposed filtermodules on a concern. When we add a logging concern, we affect this data dependency for each concern. This means that FILTH needs to recalculate filtermodule orderings for each concern. To come to this conclusion, however, INCRE needs to perform dependency checks for each concern. Hence, FILTH will always have a performance loss in the third scenario rather than a performance improvement.

   (b) **SECRET and the third scenario.** In Section C.5.1, we described that one of the data dependencies of SECRET are the filtermodule orderings calculated by FILTH. In the above explanation for FILTH, however, we concluded that the filtermodule orderings of all concerns are affected when we add a logging concern. Thus, in the third scenario, SECRET needs to re-perform semantic analysis for each concern. Again, to come to this conclusion, INCRE needs to perform dependency checks for each concern. This explains why SECRET will always have a performance loss in the third scenario rather than a performance improvement.

The next sections present and evaluate these charts one by one.

8.3 Non-incremental Compilation Time of Compilation Modules



	COP	DUM	HAR	COL	LOL	FIR	SIG	AST	REC	ILI	SEC	CON2	BAC
⬜ EX1	2%	3%	14%	19%	6%	26%	3%	3%	8%	11%	1%	3%	0%
🟥 EX2	1%	5%	8%	12%	3%	13%	7%	3%	20%	23%	0%	4%	1%
🟨 EX3	0%	5%	6%	10%	3%	3%	5%	1%	36%	27%	0%	2%	1%
🟪 EX4	0%	4%	2%	2%	1%	1%	18%	1%	12%	40%	7%	9%	3%

Figure 8.1: Share of compilation modules in non-incremental compilation time

The chart presented in Figure 8.1, shows the shares of the compilation modules in the total compilation time of four examples. As expected, we see fluctuations in consumption of compilation time by compilation module and example. From these fluctuations, we draw the following conclusions:

1. For smaller examples, the compilation modules HARVESTER and COLLECTOR mainly cover the compilation time. This means that, in order to decrease compilation time of smaller examples, it might be wise to concentrate on these two compilation modules first.
2. For larger examples, the compilation modules SIGN, RECOMA and ILICIT mainly cover the compilation time. Hence, in order to decrease compilation time of larger examples, it might be wise to concentrate on these three compilation modules first.
3. The scalability of the Compose⋆ non-incremental compiler suffers most from the compilation modules SIGN and ILICIT. These two compilation modules lose the most performance when we upscale examples. Thus, to ensure scalability of Compose⋆ compilation, these two compilation modules need improvement.

8.4 Performance Improvement by Example and Scenario

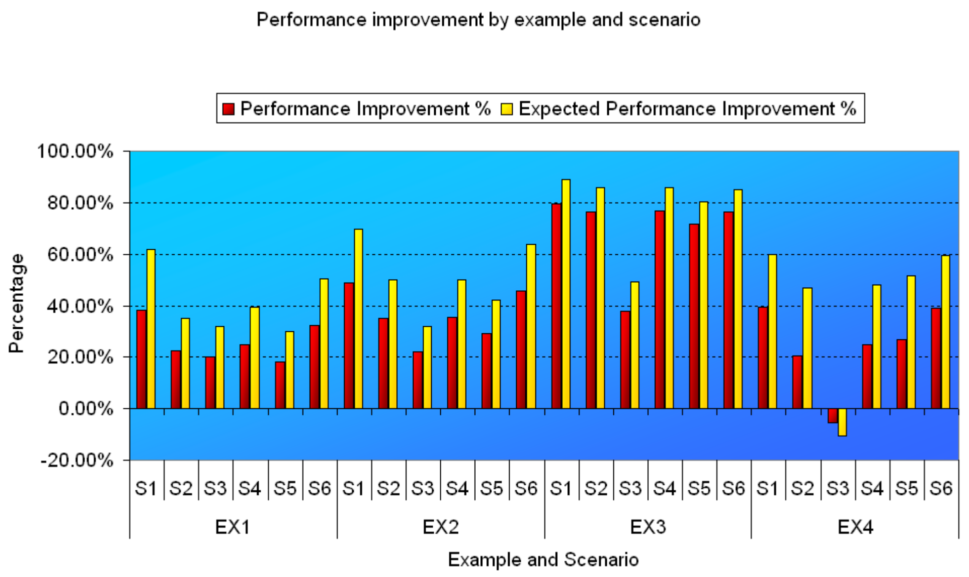


Figure 8.2: Performance improvement of incremental compiler by example and scenario

The chart presented in Figure 8.2, shows, for each test case, the performance improvement of the incremental compiler compared to the non-incremental compiler. The first bar depicts the performance improvement in percentage of the total compilation time of all compilation modules. The second bar depicts the performance improvement in percentage of the total compilation time of only incremental compilation modules. This way, we have made a distinction between the actual realized performance improvement and an expected performance improvement (when we have enhanced all compilation modules with incremental performance). In Section 8.2, we described two results which we expected to see in this chart. Below, we verify these two expectations:

1. The first expectation is that out of all scenarios, we should realize the biggest performance improvement in the first scenario. When we look at the chart, we see that this is indeed the case.
2. The second expectation is that all test cases should realize a positive performance improvement. When we look at the chart, we see that this is the case for all but except one test case. We can explain this unexpected result as follows. Our incremental rebuilding solution requires two repositories: the repository being built and the repository kept from the previous compilation of the same project (see Section 4.3). To read and write a repository, INCRE uses Java's standard serialization mechanism called Java Object Serialization (JOS). By using this mechanism, however, it cannot read parts of the repository on demand. Therefore, INCRE needs to read the whole repository at once. This big inhale results in a large number of objects allocated to memory at the start and until the end of compilation. This, in turn, increases the chance that the application runs out of memory and consequently triggers the JVM's garbage collector to free up memory. Based on experiments, we conclude that for the first three examples, the garbage collector is not (or not often) triggered. This is in contrast to the fourth example, for which INCRE needs

much more memory and triggers the garbage collector multiple times. This triggering occurs even more in the third scenario, in which the repository contains more objects compared to the other scenarios because of the extra logging concern. In other words, INCRE suffers from scalability due to the increasing influence of the garbage collector. This problem is inherent to our choice of Java's standard object serialization and can only be solved by choosing a Java object persistence technology that supports on-demand retrieval of objects.

8.5 Overhead by Example and Scenario

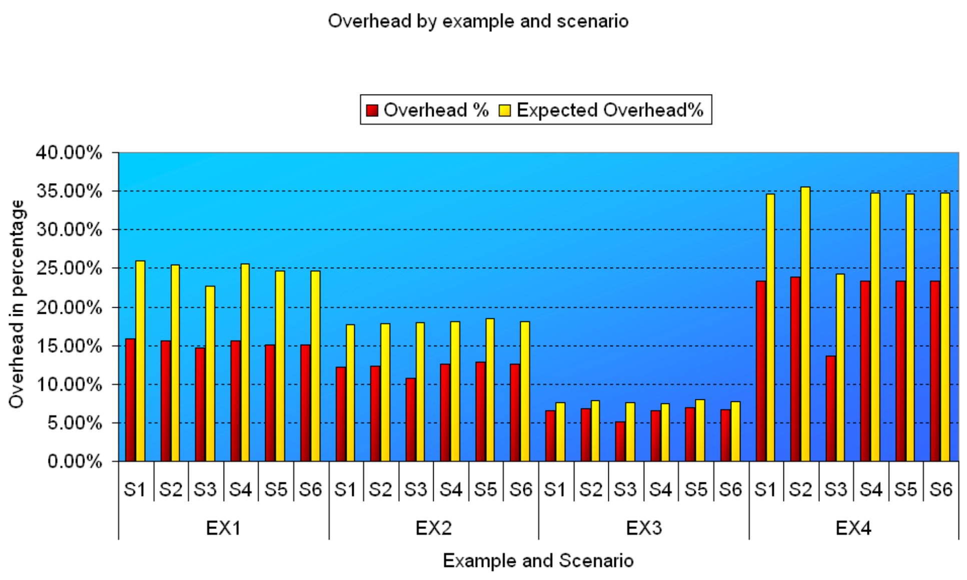
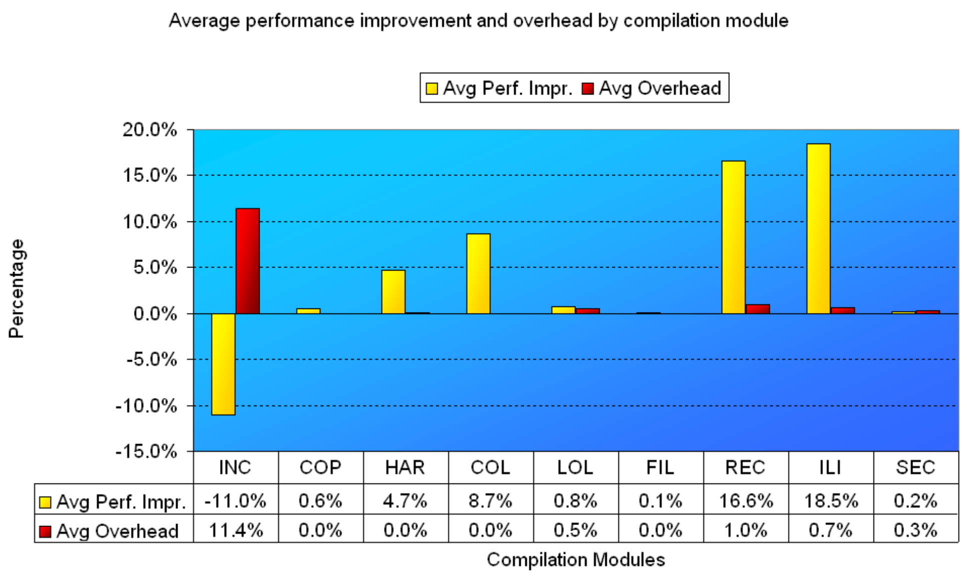


Figure 8.3: Overhead of incremental compiler by example and scenario

The chart presented in Figure 8.3, shows, for each test case, the overhead of the incremental compiler. The first bar depicts the overhead in percentage of the total compilation time of all compilation modules. The second bar depicts the overhead in percentage of the total compilation time of only incremental compilation modules. This way, we have made a distinction between the actual realized overhead and an expected overhead (when all compilation modules are enhanced with incremental performance). From this chart, we draw the following conclusions:

1. As expected, out of all scenarios, we realize the maximum overhead in the first scenario.
2. For the first three examples, the overhead tends to decrease when examples are upscaled. However, this does not apply to the fourth example, in which the overhead suddenly increases. This latter implies that INCRE suffers from scalability issues. The exact reason for this unexpected result can be found in Section 8.7.

8.6 Average Performance of Compilation Modules



	INC	COP	HAR	COL	LOL	FIL	REC	ILI	SEC
🟨 Avg Perf. Impr.	-11.0%	0.6%	4.7%	8.7%	0.8%	0.1%	16.6%	18.5%	0.2%
🟥 Avg Overhead	11.4%	0.0%	0.0%	0.0%	0.5%	0.0%	1.0%	0.7%	0.3%

Figure 8.4: Average performance improvement and overhead of compilation modules

The chart presented in Figure 8.4, shows the average performance improvement and overhead of nine compilation modules in percentage of the average compilation time of all test cases. By overhead of the incremental compilation modules, we mean compilation time consumed by INCRE to enforce incremental performance of that compilation module (e.g., data comparison). By overhead of INCRE, we mean compilation time consumed by INCRE to read and write the repository. As expected, we see fluctuations in performance improvement between compilation modules. From these fluctuations, we draw the following conclusions:

1. The compilation modules RECOMA and ILICIT profit the most from incremental compilation. These compilation modules are showing respectively a performance improvement of sixteen and eighteen percent. The other six incremental compilation modules are also showing performance improvements but in lesser degree. This means that, when it comes to maintenance of the incremental compiler, it might be wise to concentrate on the compilation modules RECOMA and ILICIT first.
2. The performance improvement realized for the eight incremental compilation modules, outweigh the average performance decrease created by INCRE. This means that the incremental compiler has a performance advantage over the non-incremental compiler (see also Section 8.4).
3. Reading and writing the repository is relatively costly compared to INCRE's other tasks. This implies that, in order to improve INCRE's performance further, it might be wise to concentrate on improving these tasks first.

8.7 Evaluation of INCRE

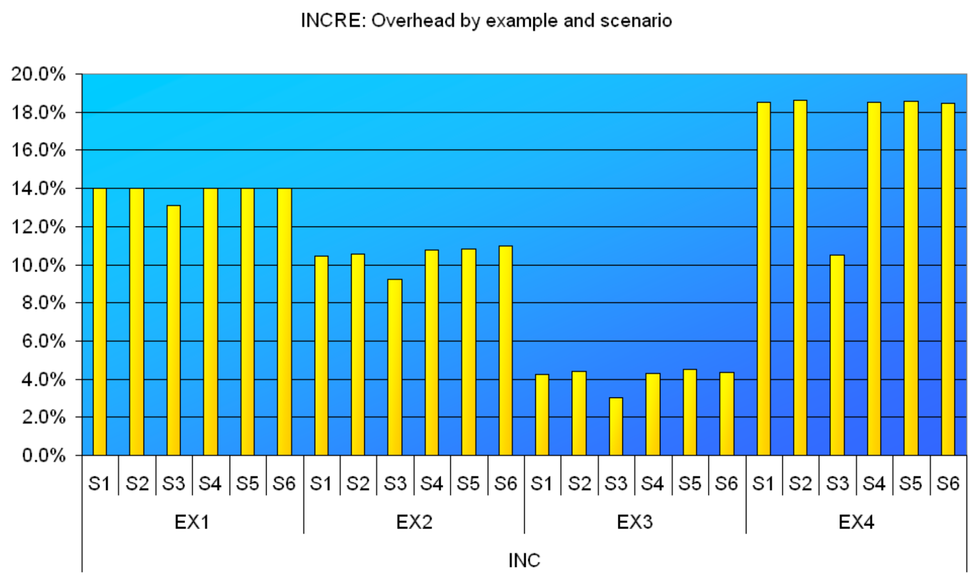


Figure 8.5: Overhead created by INCRE

The chart presented in Figure 8.5, shows the overhead of the compilation module INCRE for all test cases. By overhead of INCRE, we mean compilation time consumed by INCRE to read and write the repository. From this chart, we draw the following conclusion. For the first three examples, INCRE's overhead tends to decrease as examples are upscaled. However, this does not apply to the fourth example, for which the overhead suddenly increases. In other words, INCRE's reading and writing of the repository suffers from scalability. In Section 8.4, we already explained that this can be subscribed to the big inhale problem of Java's standard object serialization mechanism.

8.8 Evaluation of COPPER

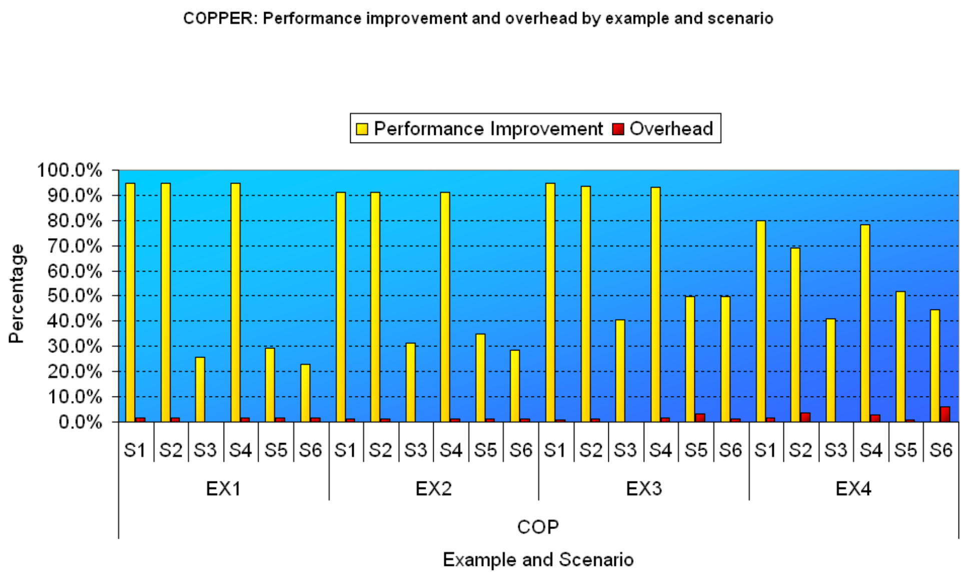


Figure 8.6: COPPER: Performance improvement and overhead by example and scenario

The chart presented in Figure 8.6, shows, for each test case, the performance improvement and overhead of COPPER in percentage of its non-incremental compilation time. From this chart, we draw the following conclusions:

1. As expected, COPPER is showing high performance improvements in every test case. This means that in general, it is beneficial to use COPPER's incremental performance rather than its non-incremental performance.
2. The maximum overhead of COPPER is minimal. This also implies that its maximum loss is minimal. This means that there is almost no risk of losing performance when you turn on COPPER's incremental performance.
3. Scenarios S1, S2 and S4 are showing the best performance improvements. This is expected and explained as follows. The incremental strategy of COPPER ensures that untouched concern files are not parsed again. Instead, COPPER performs a copy operation for these concern files. In scenarios S1, S2 and S4, we do not touch any concern file. Thus, in these scenarios, all concern files are not parsed again but instead copied. This is not the case for the other three scenarios. In these scenarios, we add or modify one concern file, which COPPER needs to parse. Together with the performance improvements of Figure 8.6, we can now conclude that COPPER's copying for a concern file is much faster than parsing that concern file. In other words, the less concern files you modify, the faster COPPER will perform its task.

8.9 Evaluation of HARVESTER

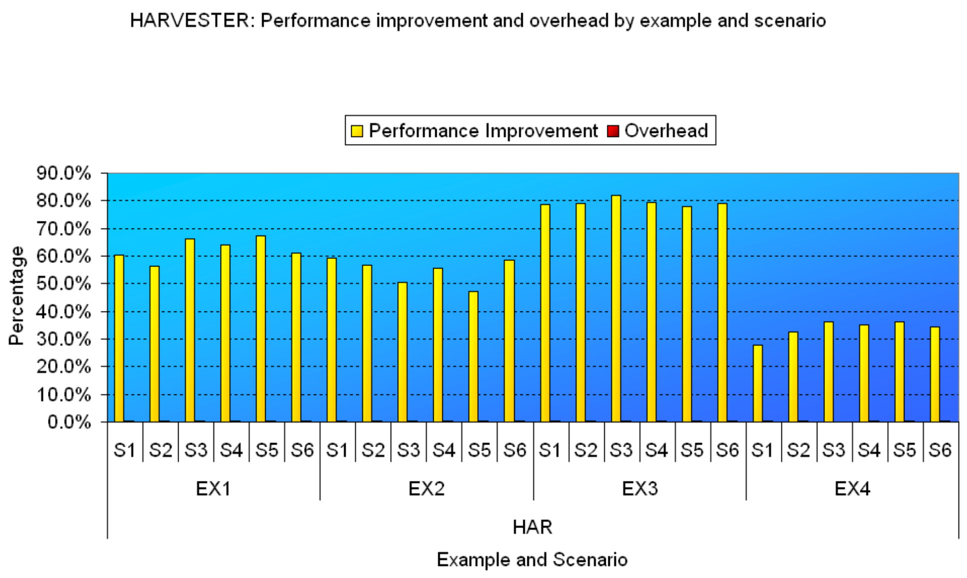


Figure 8.7: HARVESTER: Performance improvement and overhead by example and scenario

The chart presented in Figure 8.7, shows, for each test case, the performance improvement and overhead of HARVESTER in percentage of its non-incremental compilation time. From this chart, we draw the following conclusions:

1. The performance improvement varies from thirty to eighty percent. So, as expected, HARVESTER is showing high performance improvement in every test case. This means that, in general, it is more beneficial to use HARVESTER's incremental performance rather than its non-incremental performance.
2. The maximum overhead of HARVESTER is minimal. This also implies that its maximum loss is minimal. This means that there is almost no risk of losing performance when you turn on HARVESTER's incremental performance.
3. Since the performance improvement in scenario S1 is not hundred percent, we can conclude that HARVESTER is always reading one or more assemblies during incremental compilation. Looking closer to HARVESTER's process tells us that HARVESTER is always reading the .NET assembly called ”dummies.dll”. The compilation module DUMMER creates this assembly. This compilation module has not yet incremental performance, so it regenerates ”dummies.dll” in every compilation. This means that we can further improve HARVESTER by enhancing DUMMER with incremental performance.

8.10 Evaluation of COLLECTOR

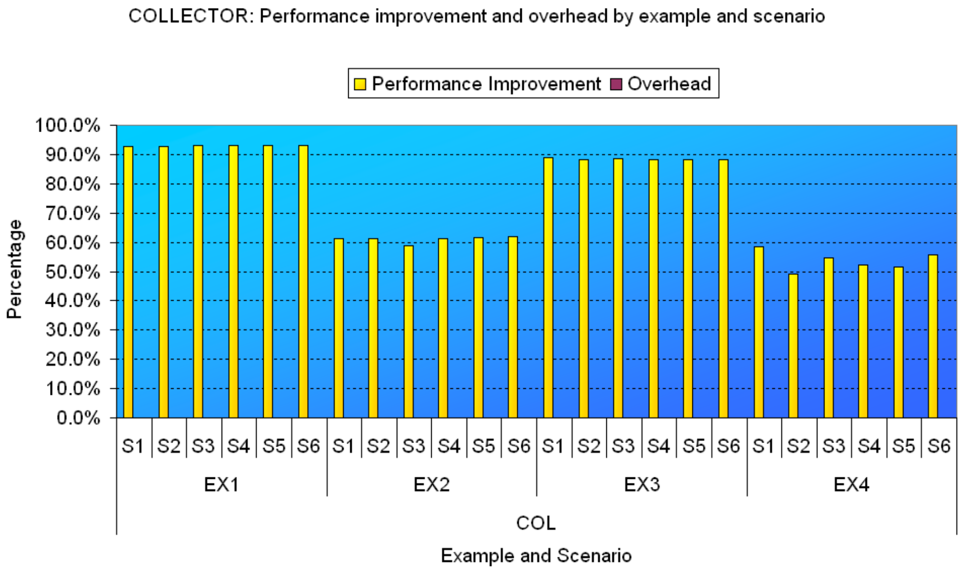


Figure 8.8: COLLECTOR: Performance improvement and overhead by example and scenario

The chart presented in Figure 8.8, shows, for each test case, the performance improvement and overhead of COLLECTOR in percentage of its non-incremental compilation time. From this chart, we draw the following conclusions:

1. As expected, COLLECTOR is showing high performance improvement in every test case. Thus, in general, it is more beneficial to use COLLECTOR's incremental performance rather than its non-incremental performance.
2. The performance improvement varies from fifty to ninety percent. These high percentages are the direct result of HARVESTER's minimization of the XML file containing type information. Because this way, COLLECTOR only needs to read a minimal XML file and copy the remaining (missing) type information from the backup repository. This copy operation is significant faster compared to the costly read operations.

8.11 Evaluation of LOLA

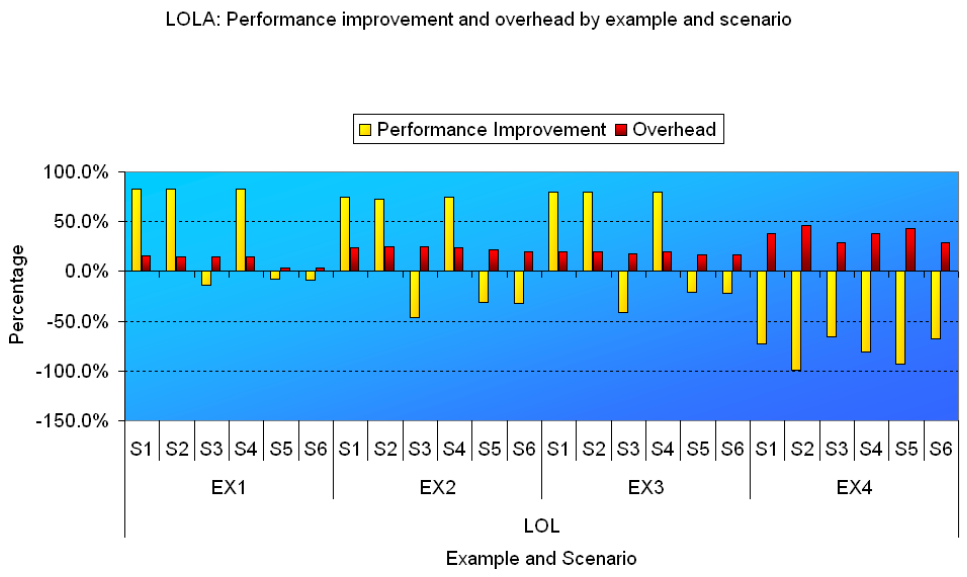


Figure 8.9: LOLA: Performance improvement and overhead by example and scenario

The chart presented in Figure 8.9, shows, for each test case, the performance improvement and overhead of LOLA in percentage of its non-incremental compilation time. From this chart, we draw the following conclusions:

1. LOLA is showing a performance loss in the fourth example because of the influence of the garbage collector (see Section 8.4). This means that it is not beneficial to turn on incremental performance of LOLA for large examples.
2. In the first three examples, LOLA has a performance loss for scenarios S3, S5 and S6. We can explain this unexpected loss as follows. LOLA consists of two phases: initialization of the prolog engine and evaluation of the prolog queries. Note that the initialization phase is only needed when LOLA needs to evaluate one or more predicate selectors. Looking at the six scenarios, tells us that we only add or modify a predicate selector in scenarios S3, S5 and S6. This means that the initialization phase is only performed in these scenarios and not in scenarios S1, S2 and S4. Together with the performance improvements of Figure 8.9, we can now conclude that LOLA's initialization phase is a relatively costly operation compared to the evaluation of prolog queries. In fact, it is so costly that the overhead of LOLA does not outweigh the performance improvement of skipping predicate queries. In other words, it is only beneficial to use the incremental performance of LOLA, when you can avoid LOLA's initialization phase. In general, this is not likely as a programmer constantly adds or modifies a predicate selector. Thus, LOLA's incremental performance needs improvement to benefit a programmer.

8.12 Evaluation of FILTH

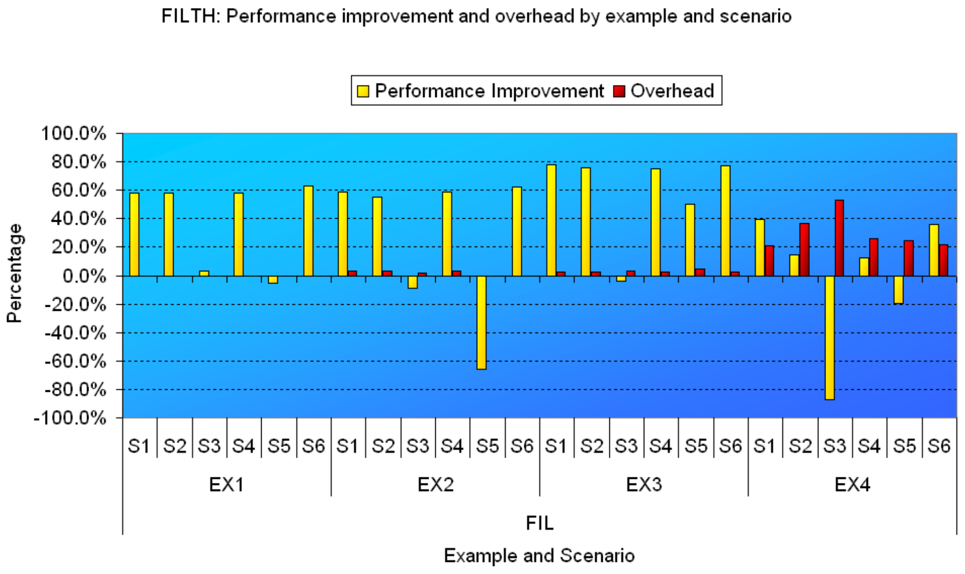


Figure 8.10: FILTH: Performance improvement and overhead by example and scenario

The chart presented in Figure 8.10, shows, for each test case, the performance improvement and overhead of FILTH in percentage of its non-incremental compilation time. From this chart, we draw the following conclusion:

1. FILTH is showing the worst performance improvements for large examples. In fact, these performance improvements are so low in absolute terms (milliseconds), that it is perhaps more beneficial to turn the incremental performance of FILTH off for large examples. This latter can be explained as follows. When FILTH is not incremental, we do not need to store its output any longer. This means that we can reduce the number of objects stored in the backup repository. This directly results in less objects allocated to memory, which reduces the chance of triggering the garbage collector to free memory (see Section 8.4). This may result in a performance improvement of the garbage collector that outweighs the small performance improvement of FILTH.

8.13 Evaluation of SECRET

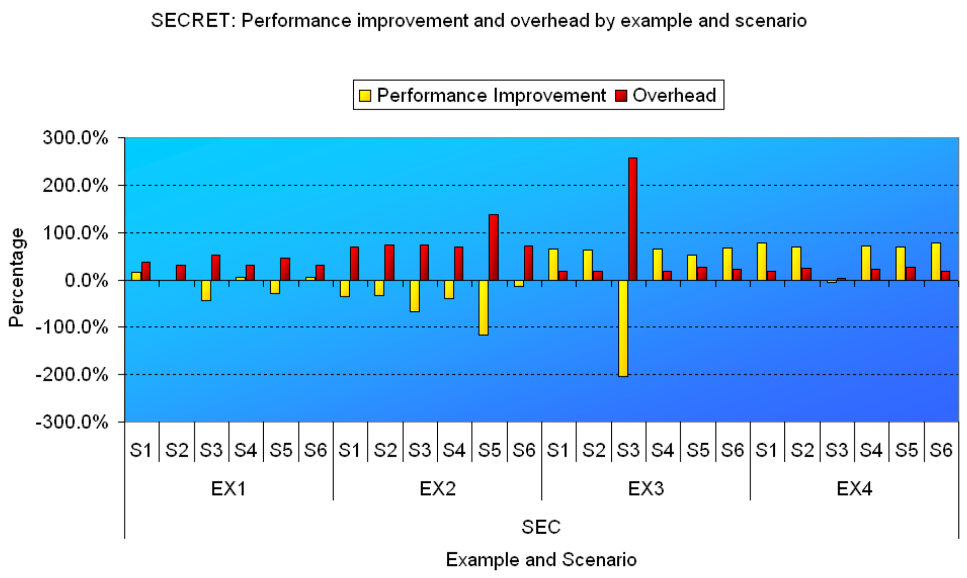


Figure 8.11: SECRET: Performance improvement and overhead by example and scenario

The chart presented in Figure 8.11, shows, for each test case, the performance improvement and overhead of SECRET in percentage of its non-incremental compilation time. From this chart, we draw the following conclusion:

1. For smaller examples, SECRET has a performance loss rather than performance improvement. The reason for this unexpected loss of performance can be found in the number of filtermodule orderings in each example. Table 8.1 lists the number of filtermodule orderings by example. From this table, we conclude that the number of filtermodule orderings significantly increases for the two largest examples. This increase has the following consequence for SECRET's execution time. For each test case, we have used SECRET's progressive mode. Because of this mode, SECRET needs to analyze all possible filtermodule orderings per concern. This means that the execution time of SECRET is linear to the number of filtermodule orderings. Thus, SECRET's execution time also significantly increases for the two largest examples. When we now look at the overhead percentages in Figure 8.11, we see that the overhead decreases when examples are upscaled. This tells us that the number of filtermodule orderings has less influence on SECRET's overhead than on SECRET's analysis. In other words, the more filtermodule orderings SECRET needs to analyze, the more it can benefit from its incremental performance.

8.14 Evaluation of RECOMA

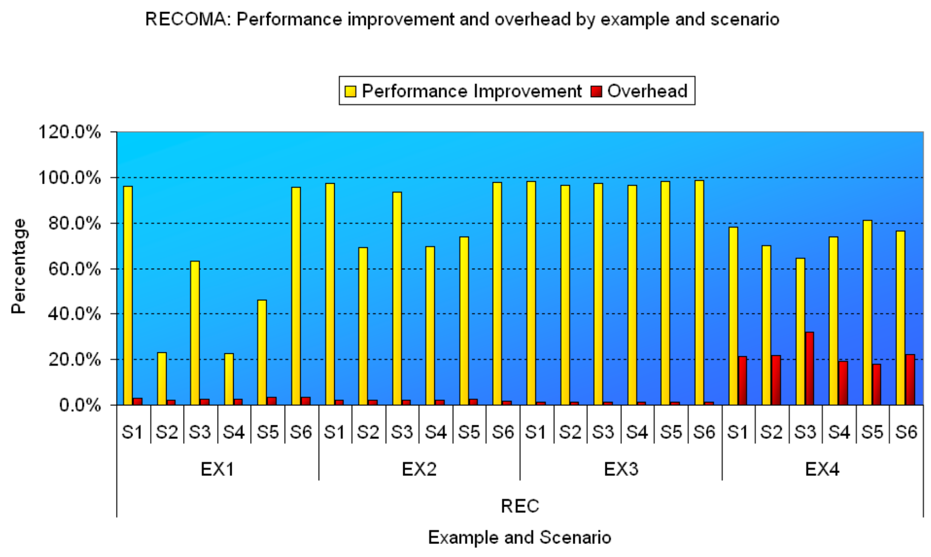


Figure 8.12: RECOMA: Performance improvement and overhead by example and scenario

The chart presented in Figure 8.12, shows, for each test case, the performance improvement and overhead of RECOMA in percentage of its non-incremental compilation time. From this chart, we draw the following conclusions:

1. The performance improvement varies from twenty to ninety-nine percent. So, as expected, RECOMA is showing high performance improvement in every test case. This means that, in general, it is more beneficial to use RECOMA's incremental performance rather than its non-incremental performance.
2. In the fourth example, RECOMA's overhead is relatively high compared to the other three examples. The reason for this unexpected high level of overhead is twofold. The first reason can be found in the influence of the garbage collector. The garbage collector has almost no influence on the first three examples. However, in the fourth example, the number of allocated objects becomes so high that it triggers the garbage collector to free memory. These garbage collections increase the overhead. The second reason can be found in the complexity of the fourth example. By complexity, we mean in this case, the degree of dependencies between source files. The complexity of the fourth example is much higher compared to the other three examples. This higher degree of complexity has the following consequence for RECOMA's overhead for a source file. When we look at RECOMA's dependencies (see Section 7.2.2), we see that two of them strongly depend on the number of relations between source files. These two are the source files referenced by a source file and the full signatures of concerns extracted from these referenced source files. Thus, the higher the complexity of a project, the more compilation time INCRE needs to check these two dependencies. Therefore, the overhead in the fourth example is higher than in the other three less complex examples.

8.15 Evaluation of ILICIT

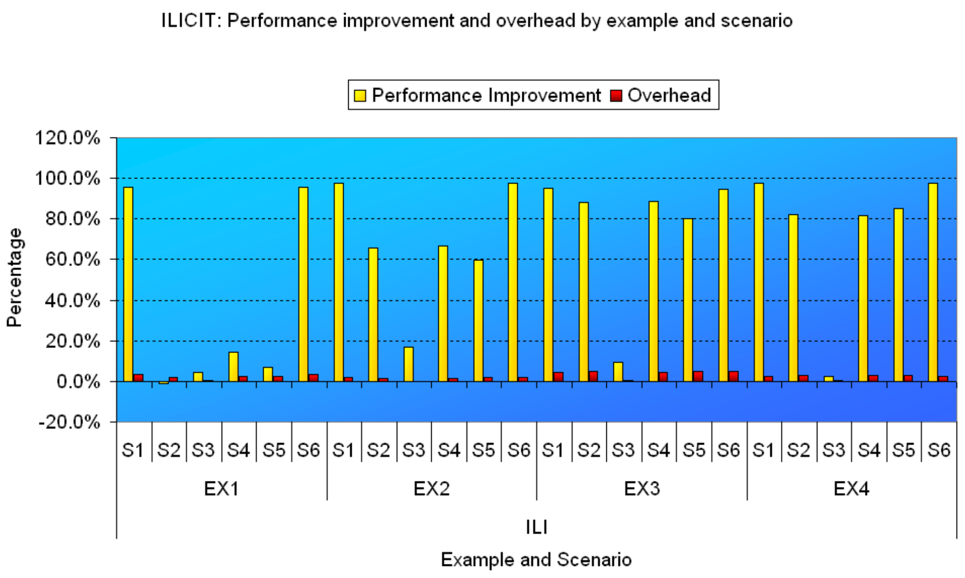


Figure 8.13: ILICIT: Performance improvement and overhead by example and scenario

The chart presented in Figure 8.13, shows, for each test case, the performance improvement and overhead of ILICIT in percentage of its non-incremental compilation time. From this chart, we draw the following conclusions:

1. ILICIT is showing high performance improvements in all test cases except for scenario S3. This is expected and explained as follows. In scenario S3, we add a logging concern that logs each method call. To enforce this logging concern at runtime, the weaver must weave some additional IL-code at each method call. In our examples, this affects almost each assembly. In other words, almost each assembly needs to be rewoven. This in contrast to the other scenarios, where the changes result in reweaving of only a small set of assemblies (S2,S4,S5) or even zero assemblies (S1,S6). Hence, ILICIT consumes much more compilation time in scenario S3 than in the other scenarios. This explains the low performance scores for scenario S3 compared to the other scenarios.
2. The maximum overhead of ILICIT is minimal. This also implies that its maximum loss is minimal. So there is almost no risk of losing performance when you turn on ILICIT's incremental performance.

8.16 Conclusions

This chapter evaluated the incremental performance of eight compilation modules by means of charts. To conclude this chapter, we here briefly summarize the main expected and unexpected results represented in these charts:

Expected Results:

1. For smaller examples, the compilation modules HARVESTER and COLLECTOR mainly cover the compilation time. Thus, in order to decrease compilation time of smaller examples, it might be wise to concentrate on these two compilation modules first.

2. For larger examples, the compilation modules SIGN, RECOMA and ILICIT mainly cover the compilation time. Thus, in order to decrease compilation time of larger examples, it might be wise to concentrate on these three compilation modules first.

3. The compilation modules RECOMA and ILICIT profit the most from incremental compilation. This means that, when it comes to maintenance of the incremental compiler, it might be wise to concentrate on the compilation modules RECOMA and ILICIT first.

4. The compilation modules COPPER, HARVESTER, COLLECTOR, RECOMA and ILICIT are showing high performance improvements in every test case. This means that, in general, it is more beneficial to use the incremental performance of these compilation modules rather than their non-incremental performance.

Unexpected Results:

1. INCRE's reading and writing of the repository suffers from scalability due to the increasing influence of the garbage collector. This problem is inherent to our choice of Java object serialization. We can only solve this by choosing a Java object persistence technology that supports on-demand retrieval of objects.

2. Reading and writing the repository is relatively costly compared to INCRE's other tasks. This implies that, in order to improve INCRE's performance further, it might be wise to concentrate on improving these tasks first.

3. We can further improve the compilation modules HARVESTER and COLLECTOR by enhancing DUMMER with incremental performance.

4. The incremental performance of the compilation modules LOLA and FILTH suffer from scalability due to the increasing influence of the garbage collector. Therefore, it is not beneficial to use the incremental performance of these compilation modules for large examples.

5. As expected the more filtermodule orderings SECRET needs to analyze, the more it can benefit from its incremental performance. However, this also unexpectedly implies that it is not beneficial to use SECRET's incremental performance for small examples.

Chapter 9

Conclusion, Future and Related Work

In this final chapter, we conclude this thesis. The chapter is organized as follows. First, we summarize and evaluate the contributions of this thesis with respect to incremental compilation in Compose⋆. Finally, we describe some future and related work on this subject.

9.1 Conclusion and Evaluation

The previous version of Compose⋆ contained sixteen compilation modules. Each of them suffered from the same problem. They did not support any kind of incrementality. This means that they performed all their operations in every compilation without taking advantage of the work already done in a previous compilation. This unavoidably resulted in compilations consisting of numerous redundant repeats of operations. These redundant operations slowed down compilation to such a degree that it became a burden for Compose⋆ programmers. Therefore, the motivation for this thesis has been to develop an incremental compiler for Compose⋆.

To achieve this goal, we proposed two possible solution approaches called *restoration* and *rebuilding*. The first approach considers the repository from a previous compilation as a starting point. During incremental compilation, each compilation module adapts this repository to reflect the correct compilation results. The second approach considers the repository kept from a previous compilation as a backup to rebuild a new repository faster. We compared these two solution approaches on four software metrics: efficiency, simplicity, severity and maintainability. From these software metrics, we concluded that the rebuilding approach is the most desirable one for Compose⋆.

Based on the rebuilding approach, we designed and implemented a new compilation module called *INCRE*. This compilation module offers incremental performance as a service to all other compilation modules. In order to have this service used by a compilation module, you must complete the following five-step procedure:

1. The first step is to acquire knowledge of the input, output and data dependencies of the compilation module. Each compilation module processes a set of input objects to some output. The way it processes its input objects depends on a set of a data. These data dependencies are unique for each combination of input object and compilation module. Knowledge of the input, output and data dependencies is essential for enhancing the compilation module with incremental performance.

2. The second step is to ensure that each repository object produced and used by the compilation module implements the `java.io.Serializable` interface or inherits that implementation from its object hierarchy. Otherwise, the serialization process will throw a `java.io.NotSerializableException`.

3. The third step is the addition of a so-called *copy* operation to the compilation module. This copy operation should copy the output belonging to a specific input object from the backup repository into the repository being built. In order to safe compilation time, this copy operation should be faster than the compilation module's original processing of an input object.

4. The fourth step is configuration of the input and data dependencies of a compilation module. This is achieved by using a XML configuration file. In this XML file, you can configure the data dependencies of a compilation in different ways by using the designed XML tags. Appendix A presents an XML Document Type Definition (DTD) [4] of the XML configuration file.

5. The final step is not required but highly advisable. For each compilation module, you can configure on which fields to compare two objects of a certain type. When you do not configure fields, INCRE compares two objects on all fields by default. Otherwise, it compares two objects on the configured fields only. In this way, you can minimize the number of comparisons by INCRE to decrease compilation time further.

9.1.1 Evaluation

To proof our concept, we have enhanced eight of the sixteen compilation modules with incremental performance. The changes to these compilation modules have been tested on four .NET projects of different sizes. For each project, we have used six different OOP and AOP scenarios. The results of these tests are described in the evaluation chapter. From these test results, we draw the following three main conclusions.

Firstly, the different test cases show performance improvements from thirty to eighty-five percent. The average performance improvement lays around fifty-five percent. This amount tells us that incremental compilation reduces the average compilation of the test cases with a factor of two compared to non-incremental compilation. In other words, on average, it is beneficial to use incremental compilation rather than non-incremental compilation.

Secondly, the compilation module INCRE has one major drawback. It suffers from scalability. The reason for this is the following. To read and write the repository, INCRE uses Java's standard serialization mechanism called Java Object Serialization (JOS). By using this mechanism, it cannot read parts of the repository on demand. Therefore, INCRE needs to read the whole repository at once. This result in a large number of objects allocated to memory at the start of a compilation. These objects stay in memory until the end of compilation. However, the more objects allocated to memory, the bigger the chance of triggering the JVM's garbage collector to free memory. These garbage collections are expensive and slow down compilation. In other words, INCRE's reading of the repository suffers from scalability due to the increasing influence of the garbage collector. This problem is inherent to our choice of Java object serialization. We can only solve this by choosing a Java object persistence technology that supports on-demand retrieval of objects.

Thirdly, the performance improvements vary by compilation module and project size. This also means that a compilation module may not benefit from incremental compilation for all kinds of projects. For instance, the compilation module SECRET only benefits from incremental compilation when the project is "large" enough. By large, we mean a project that uses a large number of filtermodules. In other words, the more filtermodules the project uses, the more SECRET benefits from incremental compilation. On the other hand, there are also compilation modules that only benefit from incremental compilation when the example is small. Examples of these kinds of compilation modules are FILTH and LOLA. Thus, depending on the size of a project, you may need to turn off incremental performance of a compilation module to achieve maximum performance improvement.

9.2 Future Work

In this section, we describe possible future work on incremental compilation in Compose⋆.

9.2.1 Automation of XML Configuration

The first area we would like to investigate is automatic generation of the XML configuration file used by INCRE. The reason for this is twofold.

Firstly, in order to enhance a compilation module with incremental performance, a Compose⋆ developer needs to configure its data dependencies in the XML configuration file. Currently, this has to be done manually. This can be a tedious and error-prone process especially when the compilation module is complex. Therefore, we would like to investigate whether it is possible to configure the data dependencies of a compilation module, or a part of it, automatically. This would relieve the Compose⋆ developer of the task of maintaining and developing incremental performance for compilation modules.

Secondly, in the previous evaluation section, we described that the incremental performance of a compilation module depends on the size of the user-provided project. There are compilation modules that perform well for small-sized projects and, on the other hand, for large-sized projects. This means that, depending on the size of a project, a programmer may need to turn off incremental performance of a compilation module to achieve fastest compilation time. Currently, a programmer can establish this by manually modifying the XML configuration file. We would like to investigate whether it is possible to automate this process.

9.2.2 Further Modulation and Enhancing with Incremental Performance

A second area of future work is further modulation of the compilation process and enhancing remaining compilation modules with incremental performance. Currently, we have enhanced eight of the sixteen compilation modules. This means that we can add incremental performance to another eight compilation modules to improve the incremental compilation further. In the evaluation chapter, we described that the compilation module DUMMER is a good candidate to start with. Enhancing this compilation module with incremental performance does not only benefit DUMMER itself, but it would also benefit the compilation modules HARVESTER and COLLECTOR. In addition, we would like to investigate whether it is possible to modulate the compilation process further. When we do this, we create more and less complex compilation modules that we can enhance with incremental performance. By doing this, we could further optimize incremental compilation in Compose⋆.

9.2.3 Alternative Java Object Persistence Mechanism

The main limitation of the developed incremental compiler is that it suffers from scalability. In the evaluation section, we described that this problem is inherent to our choice of Java object serialization as Java object persistence technology. This persistence technology does not support on demand retrieval of objects, which causes scalability problems. In the future, we would like to have an alternative persistence technology that guarantees scalability of incremental compilation in Compose⋆. Recent work shows that db4o [1], an open source object database, might be a good candidate.

9.3 Related Work

To conclude this chapter, we here briefly present two other AOP implementations that successfully integrated incremental compilation.

9.3.1 AspectJ

The original AOP compiler for the AspectJ [7] language is called *ajc*. This compiler supports *incremental* byte-code weaving on a per-class basis. This means that it minimizes the number of weave operations between classes and aspects in .class form. AspectJ recognizes the complication of aspects on incremental weaving by stating that the AspectJ tools are getting better at minimizing weaving, but to some degree, weaving is unavoidable due to the crosscutting semantics [2].

9.3.2 Apostle

Apostle is an aspect-oriented extension to the Smalltalk [51] programming language. Apostle uses *incremental* source-code weaving to weave Smalltalk source code and aspects together into equivalent pure Smalltalk results called *target models*. These models are compiled and installed by using a Smalltalk compiler. For further reading, [13] and [14] give a comprehensive description of the Apostle weaver implementation.

Bibliography

[1] db40 - An Open Source Object Database. URL `http://www.db4o.com/`.

[2] Introduction to the aspectj tools. URL `http://www.eclipse.org/aspectj/doc/released/devguide/bytecode-concepts.html`.

[3] Language and Compilers in the .net Framework. URL `http://msdn2.microsoft.com/en-us/library/ms229699.aspx`.

[4] XML DTD - An Introduction to XML Document Type Definitions. URL `http://www.xmlfiles.com/dtd/`.

[5] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, December 1990.

[6] Objectmatter: Glossary of Terms, 2003. URL `http://www.objectmatter.com/glossary.htm`.

[7] AspectJ. AspectJ. URL `http://www.eclipse.org/aspectj/`.

[8] L. Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994. URL `http://trese.cs.utwente.nl/publications/paperinfo/bergmans.phd.pi.top.htm`.

[9] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, Oct. 2001.

[10] S. R. Boschman. Performing transformations on .NET intermediate language code. Master's thesis, University of Twente, The Netherlands, Aug. 2006.

[11] R. Bosman. Automated reasoning about Composition Filters. Master's thesis, University of Twente, The Netherlands, Nov. 2004.

[12] O. Conradi. Fine-grained join point model in Compose*. Master's thesis, University of Twente, The Netherlands, Aug. 2006.

[13] B. S. de Alwis. Aspects of incremental programming, 2002. URL `http://www.cs.ubc.ca/grads/resources/thesis/May02/Brian_deAlwis.pdf`.

[14] B. S. de Alwis. Apostle: A simple incremental weaver for a dynamic aspect language, 2003. URL `http://www.cs.ubc.ca/~bsd/papers/tr-2003-16.pdf`.

[15] D. Doornenbal. Analysis and redesign of the Compose* language. Master's thesis, University of Twente, The Netherlands, Oct. 2006.

[16] P. E. A. Dürr. Detecting semantic conflicts between aspects (in Compose*). Master's thesis, University of Twente, The Netherlands, Apr. 2004.

[17] Eclipse Java Builder. Eclipse. URL http://www.eclipse.org/articles/Article-Builders/builders.html.

[18] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Comm. ACM*, 44(10): 29–32, Oct. 2001.

[19] H. Evans. Why object serialization is inappropriate for providing persistence in java, 1999. URL citeseer.ist.psu.edu/evans00why.html.

[20] S. I. Feldman. Make-a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–65, 1979. URL citeseer.ist.psu.edu/feldman79make.html.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison Wesley, 1995.

[22] M. Glandrup. Extending C++ using the concepts of composition filters. Master's thesis, University of Twente, 1995. URL http://trese.cs.utwente.nl/publications/paperinfo/glandrup.thesis.pi.top.htm.

[23] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley and Sons, 2003. ISBN 0471431044.

[24] T. Greanier. Discover the secrets of the Java Serialization API. 2000. URL http://www.javaworld.com/javaworld/jw-07-2000/jw-0714-flatten-p2.html.

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

[26] Hibernate. Hibernate, relational persistence for Java and .NET, 2006. URL http://www.hibernate.org/.

[27] F. J. B. Holljen. Compilation and type-safety in the Compose* .NET environment. Master's thesis, University of Twente, The Netherlands, May 2004.

[28] R. L. R. Huisman. Debugging Composition Filters. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[29] R. Hustead. Mapping XML to Java: Employ the SAX API to Map XML Documents To Java Objects. 2000. URL http://java.sun.com/developer/technicalArticles/xml/mapping/index.html.

[30] S. H. G. Huttenhuis. Patterns within aspect orientation. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[31] IBM VisualAge C++. IBM VisualAge C++. URL http://www-306.ibm.com/software/awdtools/vacpp/.

[32] JDBC. Java Database Connectivity (JDBC) Technology, 2006. URL http://java.sun.com/products/jdbc/.

[33] JDBC Drivers. JDBC Drivers, 2006. URL http://developers.sun.com/product/jdbc/drivers.

[34] JDK 1.4.2 API Class Object. JDK 1.4.2 API Class Object, 2006. URL http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html.

[35] JDO. Java Data Objects (JDO), 2006. URL http://java.sun.com/products/jdo/.

[36] JDO Portability Guidelines. JDO Portability Guidelines. URL http://www.solarmetric.com/Software/Documentation/jdospec/jdo19.html.

[37] Jikes. Jikes. URL http://jikes.sourceforge.net/.

[38] D. Jordan. JDO Links. URL http://www.jdocentral.com/JDO_Links_Body.html.

[39] D. Jordan. A Comparison Between Java Data Objects (JDO), Serialization and JDBC for Java Persistence. 2002. URL http://www.jdocentral.com/pdf/DavidJordan_JDOversion_12Mar02.pdf.

[40] M. Jordan. A Comparative Study of Persistence Mechanisms for the Java Platform. 2004. URL http://research.sun.com/techrep/2004/smli_tr-2004-136.pdf.

[41] M. Karasick. The architecture of montana: an open and extensible programming environment with an incremental c++ compiler. pages 131–142, 1998. URL http://doi.acm.org/10.1145/288195.288284.

[42] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[43] P. Koopmans. Sina user's guide and reference manual. Technical report, Dept. of Computer Science, University of Twente, 1995. URL http://trese.cs.utwente.nl/publications/paperinfo/sinaUserguide.pi.top.htm.

[44] L. R. Nackman. CodeStore and Incremental C++. *Dr Dobb's Journal*, 1997. URL http://www.ddj.com/184410345.

[45] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. PhD thesis, University of Twente, The Netherlands, June 2006.

[46] Oracle Toplink. Oracle Toplink, 2006. URL http://www.oracle.com/technology/products/ias/toplink/index.html.

[47] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the Hyperspace approach. In M. Akşit, editor, *Software Architectures and Component Technology*. Kluwer Academic Publishers, 2001. ISBN 0-7923-7576-9.

[48] A. Popovici, G. Alonso, and T. Gross. Just in time aspects. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 100–109. ACM Press, Mar. 2003.

[49] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In G. Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 141–147. ACM Press, Apr. 2002.

[50] P. Salinas. Adding systemic crosscutting and super-imposition to Composition Filters. Master's thesis, Vrije Universiteit Brussel, Aug. 2001.

[51] Smalltalk. Smalltalk. URL `http://www.smalltalk.org/main/`.

[52] T. Staijen. Towards safe advice: Semantic analysis of advice types in Compose*. Master's thesis, University of Twente, Apr. 2005.

[53] I. Sun Microsystems. Java Object Serialization. 2002. URL `http://java.sun.com/j2se/1.4.2/docs/guide/serialization/index.html`.

[54] P. Tarr, H. Ossher, S. M. Sutton, Jr., and W. Harrison. N degrees of separation: Multi-dimensional separation of concerns. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 37–61. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.

[55] J. W. te Winkel. Bringing Composition Filters to C. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[56] W. F. Tichy. Smart recompilation. *ACM Trans. Program. Lang. Syst.*, 8(3):273–291, 1986. ISSN 0164-0925. URL `http://doi.acm.org/10.1145/5956.5959`.

[57] Transparent Persistence. Transparent Persistence. URL `http://www.solarmetric.com/Software/Documentation/2.3.0/jdo_overview_intro_transpers.html`.

[58] M. D. W. van Oudheusden. Automatic Derivation of Semantic Properties in .NET. Master's thesis, University of Twente, The Netherlands, Aug. 2006.

[59] C. Vinkes. Superimposition in the Composition Filters model. Master's thesis, University of Twente, The Netherlands, Oct. 2004.

[60] D. A. Watt. *Programming language concepts and paradigms*. Prentice Hall, 1990.

[61] J. C. Wichman. The development of a preprocessor to facilitate composition filters in the Java language. Master's thesis, University of Twente, 1999. URL `http://trese.cs.utwente.nl/publications/paperinfo/wichman.thesis.pi.top.htm`.

Appendix A

INCREConfig.xml Definition

This XML file is used to configure the incremental compilation. The XML file supports the following configurations:

```
<!ELEMENT module (dependency* , comparison*)
<!ATTLIST module
              name CDATA #REQUIRED
              fulltype CDATA #REQUIRED
              input CDATA #REQUIRED
              enabled (true | false) false
              incremental (true | false) false
              summary CDATA #REQUIRED
              NB. The order of compilation modules is the control flow of compilation.
```

Description of *module* attributes:

- **name** - the name of the Compose⋆ compilation module like *"LOLA"*.
- **fulltype** - the fully qualified name of a compilation module like *"Composestar.DotNET.LOLA.DotNETLOLA"*
- **input** - defines the full type of the input of a compilation module for validation purpose. For example *"java.lang.String"*.
- **enabled** - defines whether the compilation module is enabled or not. By default *false*.
- **incremental** - defines whether the compilation module is incremental or not. By default *false*.
- **summary** - a small description of the compilation module printed before running it.

```
<!ELEMENT dependency (dependencypath)
<!ATTLIST dependency
              type ("FILE" | "OBJECT")
              name CDATA #REQUIRED
              isAdded (true | false) true
              store (true | false) false
              lookup (true | false) false
```

Description of *dependency* attributes:

- **type** - defines the type of the dependency. Either "FILE" or "OBJECT".
- **name** - defines the name of the dependency for debug purpose.
- **isAdded** - whether to check a file for 'added to project' or not. This attribute only applies to file dependencies and is by default *true*. Set this attribute to *false* to skip the check. Used for optimization purpose.
- **store** - whether to store the result of a data comparison or not. By default *false*. Used for optimization purpose.
- **lookup** - whether to lookup the result of a previous data comparison instead of comparing the data dependency. By default *false*. When this attribute is set to *true*, the dependency is only checked once for all input objects of a compilation module. Used for optimization purpose.

```
<!ELEMENT dependencypath (dependencynode)*
<!ATTLIST dependencypath EMPTY

<!ELEMENT dependencynode EMPTY
<!ATTLIST dependencynode
          type ("CONFIG" | "DYNAMIC" | " FIELD" | "METHOD")
          value CDATA #REQUIRED
```

Description of *dependencynode* attributes:

- **type** - defines the type of node. The different nodes are: confignode representing a project configuration, dynamicnode representing a dynamic object, fieldnode representing a field and methodnode representing a method.
- **value** - the value of the node. This value is a reference to a specific project configuration, dynamic object, field or method.

```
<!ELEMENT comparison (comparisontype*)
<!ATTLIST comparison EMPTY

<!ELEMENT comparisontype (comparisonfield , comparisonpath)
<!ATTLIST comparisontype
          fullname CDATA #REQUIRED
```

Description of *comparisontype* attributes:

- **fullname** - The fully qualified name of the type checked by the comparator like *"Composestar.Core.CpsProgramRepository.CpsConcern.Filtermodules.FilterModule"*

```
<!ELEMENT comparisonfield EMPTY
<!ATTLIST comparisonfield
          name CDATA #REQUIRED
```

Description of *comparisonfield* attributes:

- **name** - the name of a field.

```
<!ELEMENT comparisonpath (comparisonmethod+)
<!ATTLIST comparisonpath EMPTY

<!ELEMENT comparisonmethod EMPTY
<!ATTLIST comparisonmethod
                name CDATA #REQUIRED
```

Description of *comparisonmethod* attributes:

- **name** - the name of a method.

Appendix B

XML Configuration of Compose⋆ Compilation Modules

B.1 ASTRA Configuration

```
<module name="ASTRA" fulltype="Composestar.DotNET.TYM.SignatureTransformer.
    DotNETSignatureTransformer" enabled="true" incremental="false" summary="">
  <dependencies></dependencies>
</module>
```

Listing B.1: XML configuration of ASTRA

B.2 BACO Configuration

```
<module name="BACO" fulltype="Composestar.DotNET.BACO.DotNETBACO" enabled="true"
    incremental="false" summary="Copying assemblies..." >
  <dependencies></dependencies>
</module>
```

Listing B.2: XML configuration of BACO

B.3 CHKREP Configuration

```
<module name="CHKREP" fulltype="Composestar.Core.CHKREP.Main" enabled="true"
    incremental="false" summary="Checking repository..." >
  <dependencies></dependencies>
</module>
```

Listing B.3: XML configuration of CHKREP

B.4 COLLECTOR Configuration

```
<module name="COLLECTOR" fulltype="Composestar.DotNET.TYM.TypeCollector.
    DotNETCollectorRunner" enabled="true" incremental="false" summary="Collecting
    type information..." >
  <dependencies></dependencies>
</module>
```

Listing B.4: XML configuration of COLLECTOR

B.5 CONE Configuration

```
<module name="CONE" fulltype="Composestar.DotNET.CONE.DotNETCONE" enabled="true"
    incremental="false" summary="">
  <dependencies></dependencies>
</module>
```

Listing B.5: XML configuration of CONE

B.6 COPPER Configuration

```
<module name="COPPER" fulltype="Composestar.Core.COPPER.COPPER" input="Composestar.
    Core.Master.Config.ConcernSource" enabled="true" incremental="true" summary="
    Parsing concerns..." >
  <dependencies>
    <dependency type="FILE" name="cpsfile">
      <path>
        <node type="FIELD" nodevalue="fileName"></node>
      </path>
    </dependency>
  </dependencies>
</module>
```

Listing B.6: XML configuration of COPPER

B.7 DUMMER Configuration

```
<module name="DUMMER" fulltype="Composestar.Core.DUMMER.DummyManager" enabled="true"
     incremental="false" summary="Generating dummy sources..." >
  <dependencies></dependencies>
</module>
```

Listing B.7: XML configuration of DUMMER

B.8 FILTH Configuration

```
<module name="FILTH" fulltype="Composestar.Core.FILTH.FILTH" input="Composestar.Core
    .CpsProgramRepository.Concern" enabled="true" incremental="true" summary="
    Determining order of filter modules at shared join points..." >
  <dependencies>
    <dependency type="FILE" name="specfile">
      <path>
        <node type="CONFIG" nodevalue="FILTH_INPUT"></node>
      </path>
    </dependency>
    <dependency type="OBJECT" name="fmodules">
      <path>
        <node type="DYNAMIC" nodevalue="superImpInfo"></node>
        <node type="FIELD" nodevalue="theFmSIinfo"></node>
      </path>
    </dependency>
  </dependencies>
  <comparisons>
    <type fullname="Composestar.Core.CpsProgramRepository.CpsConcern.References.
        FilterModuleReference">
      <field name="name"></field>
    </type>
  </comparisons>
</module>
```

Listing B.8: XML configuration of FILTH

B.9 HARVESTER Configuration

```
<module name="HARVESTER" fulltype="Composestar.DotNET.TYM.TypeHarvester.
    DotNETHarvestRunner" input="java.lang.String" enabled="true" incremental="true"
    summary="Harvesting type information..." >
  <dependencies>
    <dependency type="FILE" name="assembly"></dependency>
    <dependency type="OBJECT" name="previousinput">
      <path>
        <node type="METHOD" nodevalue="Composestar.DotNET.TYM.TypeHarvester.
            DotNETHarvestRunner.prevInput"></node>
      </path>
    </dependency>
    <dependency type="FILE" name="previousassemblies">
      <path>
        <node type="METHOD" nodevalue="Composestar.DotNET.TYM.TypeHarvester.
            DotNETHarvestRunner.prevAssemblies"></node>
      </path>
    </dependency>
    <dependency type="FILE" name="externalassemblies" isAdded="false">
      <path>
        <node type="METHOD" nodevalue="Composestar.DotNET.TYM.TypeHarvester.
            DotNETHarvestRunner.externalAssemblies"></node>
      </path>
    </dependency>
  </dependencies>
</module>
```

Listing B.9: XML configuration of HARVESTER

B.10 ILICIT Configuration

```
<module name="ILICIT" fulltype="Composestar.DotNET.ILICIT.ILICIT" input="java.lang.
    String" enabled="true" incremental="true" summary="Weaving assemblies..." >
  <dependencies>
    <dependency type="FILE" name="assembly" isAdded="false"></dependency>
    <dependency type="OBJECT" name="ApplicationInfo" store="true" lookup="true">
      <path>
        <node type="CONFIG" nodevalue="ApplicationStart"></node>
      </path>
    </dependency>
    <dependency type="OBJECT" name="RunDebugLevel" store="true" lookup="true">
      <path>
        <node type="CONFIG" nodevalue="RunDebugLevel"></node>
      </path>
    </dependency>
    <dependency type="OBJECT" name="ConcernsWithFMO">
      <path>
        <node type="METHOD" nodevalue="Composestar.DotNET.ILICIT.ILICIT.
            getConcernsWithFMO"></node>
      </path>
    </dependency>
    <dependency type="OBJECT" name="CastingInterceptions">
      <path>
        <node type="METHOD" nodevalue="Composestar.DotNET.ILICIT.ILICIT.
            castingInterceptions"></node>
      </path>
    </dependency>
    <dependency type="OBJECT" name="InstantiationClasses">
      <path>
        <node type="METHOD" nodevalue="Composestar.DotNET.ILICIT.ILICIT.
            getAfterInstantiationClasses"></node>
      </path>
    </dependency>
    <dependency type="OBJECT" name="ConcernsWithOutputFilters">
      <path>
        <node type="METHOD" nodevalue="Composestar.DotNET.ILICIT.ILICIT.
            getConcernsWithOutputFilters"></node>
      </path>
    </dependency>
  </dependencies>
</module>
```

Listing B.10: XML configuration of ILICIT

B.11 LOLA Configuration

```
<module name="LOLA" fulltype="Composestar.DotNET.LOLA.DotNETLOLA" enabled="true"
    incremental="true" summary="Evaluating predicate selectors..." >
  <dependencies></dependencies>
  <comparisons>
    <type fullname="Composestar.Core.CpsProgramRepository.PrimitiveConcern">
      <field name="platformRepr"></field>
    </type>
    <type fullname="Composestar.Core.LAMA.UnitResult">
      <field name="multiRes"></field>
      <field name="singleRes"></field>
    </type>
  </comparisons>
</module>
```

Listing B.11: XML configuration of LOLA

B.12 RECOMA Configuration

```
<module name="RECOMA" fulltype="Composestar.Core.TYM.SrcCompiler.RealSourceManager"
    input="Composestar.Core.Master.Config.Source" enabled="true" incremental="true"
    summary="Compiling all sources..." >
  <dependencies>
    <dependency type="FILE" name="source">
      <path>
        <node type="FIELD" nodevalue="fileName"></node>
      </path>
    </dependency>
    <dependency type="FILE" name="xternals">
      <path>
        <node type="METHOD" nodevalue="Composestar.DotNET.COMP.DotNETCompiler.
            externalSources"></node>
      </path>
    </dependency>
    <dependency type="FILE" name="CompileLibsDependencies">
      <path>
        <node type="CONFIG" nodevalue="Dependencies"></node>
      </path>
    </dependency>
    <dependency type="OBJECT" name="fullsignatures">
      <path>
        <node type="METHOD" nodevalue="Composestar.DotNET.COMP.DotNETCompiler.
            fullSignatures"></node>
      </path>
    </dependency>
  </dependencies>
  <comparisons>
    <type fullname="Composestar.Core.CpsProgramRepository.Signature">
      <field name="methodByName"></field>
    </type>
    <type fullname="Composestar.Core.CpsProgramRepository.MethodWrapper">
      <field name="RelationType"></field>
    </type>
  </comparisons>
</module>
```

Listing B.12: XML configuration of RECOMA

B.13 REXREF Configuration

```
<module name="REXREF" fulltype="Composestar.Core.REXREF.Main" enabled="true"
    incremental="false" summary="Resolving references..." >
  <dependencies></dependencies>
</module>
```

Listing B.13: XML configuration of REXREF

B.14 SANE Configuration

```
<module name="SANE" fulltype="Composestar.Core.SANE.SANE" enabled="true" incremental
    ="false" summary="Resolving superimpositions..." >
  <dependencies></dependencies>
</module>
```

Listing B.14: XML configuration of SANE

B.15 SECRET Configuration

```
<module name="SECRET" fulltype="Composestar.Core.CKRET.SECRET" input="Composestar.
    Core.CpsProgramRepository.Concern" enabled="true" incremental="true" summary="
    Checking for semantic conflicts among aspects..." >
  <dependencies>
    <dependency type="FILE" name="SECRETConfigFile" isAdded="false">
      <path>
        <node type="CONFIG" nodevalue="SECRETConfigFile"></node>
      </path>
    </dependency>
    <dependency type="OBJECT" name="SECRETMode" store="true" lookup="true">
```

```
      <path>
        <node type="CONFIG" nodevalue="SECRETMode"></node>
      </path>
    </dependency>
    <dependency type="OBJECT" name="semantics" store="true" lookup="true">
      <path>
        <node type="METHOD" nodevalue="Composestar.Core.CKRET.SECRET.
            getSemanticAnnotations"></node>
      </path>
    </dependency>
    <dependency type="OBJECT" name="filtermoduleorders">
      <path>
        <node type="DYNAMIC" nodevalue="FilterModuleOrders"></node>
      </path>
    </dependency>
  </dependencies>
  <comparisons>
    <type fullname="Composestar.Core.CpsProgramRepository.CpsConcern.References.
        FilterModuleReference">
      <field name="name"></field>
      <field name="ref"></field>
    </type>
    <type fullname="Composestar.Core.CpsProgramRepository.CpsConcern.Filtermodules.
        FilterModule">
      <field name="inputFilters"></field>
    </type>
    <type fullname="Composestar.Core.CpsProgramRepository.CpsConcern.Filtermodules.
        Filter">
      <field name="filterElements"></field>
    </type>
    <type fullname="Composestar.Core.CpsProgramRepository.CpsConcern.Filtermodules.
        FilterElement">
      <field name="conditionPart"></field>
      <field name="matchingPatterns"></field>
    </type>
    <type fullname="Composestar.Core.CpsProgramRepository.CpsConcern.Filtermodules.
        ConditionLiteral">
      <path>
        <method name="getCondition"></method>
        <method name="getRef"></method>
        <method name="getQualifiedName"></method>
      </path>
    </type>
    <type fullname="Composestar.Core.CpsProgramRepository.CpsConcern.Filtermodules.
        Or">
      <field name="left"></field>
      <field name="right"></field>
    </type>
    <type fullname="Composestar.Core.CpsProgramRepository.CpsConcern.Filtermodules.
        MatchingPart">
      <path>
        <method name="getTarget"></method>
        <method name="getName"></method>
      </path>
      <path>
        <method name="getSelector"></method>
        <method name="getName"></method>
      </path>
    </type>
    <type fullname="Composestar.Core.CpsProgramRepository.CpsConcern.Filtermodules.
        SubstitutionPart">
      <path>
        <method name="getTarget"></method>
        <method name="getName"></method>
      </path>
      <path>
        <method name="getSelector"></method>
        <method name="getName"></method>
      </path>
    </type>
    <type fullname="Composestar.Core.CpsProgramRepository.CpsConcern.Filtermodules.
        MessageSelector">
      <field name="typeList"></field>
    </type>
    <type fullname="Composestar.DotNET.LAMA.DotNETAttribute">
      <field name="value"></field>
      <field name="target"></field>
    </type>
    <type fullname="Composestar.DotNET.LAMA.DotNETMethodInfo">
      <field name="Name"></field>
    </type>
  </comparisons>
</module>
```

Listing B.15: XML configuration of SECRET

B.16 SIGN Configuration

```
<module name="SIGN" fulltype="Composestar.Core.SIGN2.Sign" enabled="true"
    incremental="false" summary="Generating signatures..." >
  <dependencies></dependencies>
</module>
```

Listing B.16: XML configuration of SIGN

Appendix C

Realization of Incremental Compose⋆ Compilation Modules

Chapter 5 concluded with a five-step procedure for enhancing Compose⋆ compilation modules with incremental performance. In Chapter 7, we performed this procedure for the compilation modules FILTH and RECOMA. In this appendix, we perform this procedure for six other compilation modules. Note that we omit the trivial serialization step (second step).

C.1 Realization of Incremental COPPER

C.1.1 Analysis

Input: The files that contain declarations of Compose⋆ concerns (files with .cps extension).

Output: The parsed concern data are converted into Java objects and inserted into a repository.

Processing: COPPER is the parser of the Compose⋆ concern files. It scans each input file and checks whether they fulfill the Compose⋆ grammar or not.

Dependencies: The only dependent data is the structural or meaningful content of the parsed file. This is, in this case, any content except for meaningless content like tabs, spaces and comments. Note that we recognize the Compose⋆ grammar as fixed or not modifiable by the user without installing an updated Compose⋆ version. Therefore, the Compose⋆ grammar is not a dependency.

Motivation Incremental Performance COPPER parses all files in every compilation. This strategy is inefficient as it leads to redundant repeats of parsings. By applying our incremental rebuilding solution to COPPER, we should minimize these redundant parsings and improve COPPER's parsing performance.

C.1.2 Copy Operation

Each repository entity extracted from a .cps file, has an attribute called `descriptionFileName`. This attribute tells COPPER from which file a repository entity has been extracted. COPPER's copy operation uses this information to search the backup repository for all objects extracted from a skipped .cps file. Each object found this way, is cloned and copied into the repository being built.

C.1.3 Dependency Configuration

From COPPER's analysis, we conclude that COPPER has only one data dependency namely the content of the parsed file. Listing C.1 presents a configuration of this file dependency.

```
1 <dependency type="FILE" name="cpsfile">
2   <path>
3     <node type="FIELD" nodevalue="fileName"></node>
4   </path>
5 </dependency>
```

Listing C.1: Dependency Configuration of COPPER

This dependency is of type FILE and is called "cpsfile" (line 1). To tell whether a file has already been parsed in a previous compilation, we only need the actual filename of that file. This filename is referenced by the field "fileName" of a COPPER's input object. To retrieve this filename, we only need to specify a path consisting of one field node (line 3).

C.1.4 Comparison Configuration

We do not need to configure fields-restrictions for COPPER. This is because these fields-restrictions are only used for comparing repository objects and not for files (see Figure 6.9). COPPER has only one dependent data and that one is a file.

C.2 Realization of Incremental HARVESTER

C.2.1 Analysis

Input: Compiled dummy sources and assemblies referenced by the user-provided project(s).
Output: Meta-information written to an XML.
Processing: HARVESTER processes each input assembly one by one. While processing, it uses reflection to extract type information from a processed assembly. Each type directly extracted from the assembly is written to an output XML file. All indirect or referenced types are added to a *pending list*. This pending list grows and shrinks while HARVESTER processes its input assembly. HARVESTER processes this pending list of types when all types of the input assembly have been processed. Finally, it contains a list of processed types to avoid duplicates in the output XML file.
Dependencies: The following three types of data influence HARVESTER's processing of one input assembly:

- Content of the input assembly. This dependency is trivial because changing the content of an assembly results in a different output XML file.
- Content of all assemblies referenced by the input assembly. This is a dependency because it may influence the above-mentioned pending list of the input assembly. Note that not all content of referenced assemblies influence the pending list but merely the part that is actually referenced. However, you cannot tell which part without reading or reflecting the input assembly first. But, we would like to skip these read operations. Therefore, we recognize all content of referenced assemblies as one dependency instead of their actual referenced parts.

- The order of all previous (before the input assembly) processed assemblies. HARVESTER maintains a list of processed types to avoid duplicates types in the XML file. Besides this, each written type contains a *fromDLL* attribute which defines the assembly where the type is extracted from. When we reorder the processing of assemblies, we possibly influence the list of processed types and consequently the *fromDLL* attribute of a written type. Thus, for each input assembly, the order of previous processed assemblies is a dependency.

Motivation Incremental Performance HARVESTER recreates its whole XML file in every compilation. This writing strategy is inefficient as it leads to redundant repeats of writing. By applying our incremental rebuilding solution to HARVESTER, we should minimize these redundant writings and improve HARVESTER's writing performance.

C.2.2 Copy Operation

For each skipped assembly, HARVESTER performs a copy operation. This copy operation adds the assembly to a list of skipped assemblies. When all skipped assemblies have been added, the list is made available to COLLECTOR by adding it to the repository.

C.2.3 Dependency Configuration

From HARVESTER's analysis, we conclude that there are three data types that influence HARVESTER's processing of an assembly. This means that when none of these three data types have been changed since a previous compilation, HARVESTER will reproduce the same results for that assembly. Listing C.2 presents a configuration of these three data dependencies.

```
1  <dependencies>
2  <dependency type=”FILE” name=”assembly”></dependency>
3  <dependency type=”FILE” name=”previousassemblies”>
4    <path>
5      <node type=”METHOD” nodevalue=”Composestar.DotNET.TYM.TypeHarvester.
          DotNETHarvestRunner.prevAssemblies”></>
6    </path>
7  </dependency>
8  <dependency type=”FILE” name=”externalassemblies” isAdded=”false”>
9    <path>
10     <node type=”METHOD” nodevalue=”Composestar.DotNET.TYM.TypeHarvester.
          DotNETHarvestRunner.externalAssemblies”></>
11   </path>
12 </dependency>
13 </dependencies>
```

Listing C.2: Dependency Configuration of HARVESTER

The first dependency refers to the input object itself which is the filename of the to be written assembly. This dependency is of type FILE and is called ”assembly” (line 2). The second dependency's refers to the filenames of assemblies that are processed before the input assembly. These filenames are obtained by a new helper method called ”prevAssemblies” (line 5). The last dependency gives us all referenced assemblies of an input assembly. These assemblies are obtained by a new helper method called ”externalAssemblies” (line 10).

C.2.4 Comparison Configuration

We do not need to configure fields-restrictions for HARVESTER. The reason for this is the same as for COPPER. Fields-restrictions are only used for optimizing comparison of repository objects. HARVESTER does not have any object dependencies.

C.3 Realization of Incremental COLLECTOR

C.3.1 Analysis

Input: XML file produced by HARVESTER.

Output: Meta-information put into repository.

Processing: COLLECTOR reads the XML file produced by HARVESTER. It converts each found type, field, parameter and method into Java objects, registers them in a special dictionary and stores them into a repository.

Dependencies: COLLECTOR has only one trivial dependent data namely the structural or meaningful content of the input XML file.

Motivation Incremental Performance COLLECTOR reads the XML file created by HARVESTER, maps the read type information to Java objects, registers them in a special dictionary and adds the objects to the repository. This is a costly operation because the XML file is generally of megabyte size. By applying our incremental rebuilding solution to COLLECTOR, we should minimize the XML file and improve COLLECTOR's reading performance.

C.3.2 Copy Operation

The copy operation of COLLECTOR first asks the HARVESTER for all assemblies whose type information were not written to XML. Then, it searches the backup repository for all type information extracted from these assemblies. Finally, it clones, registers and adds the found type information to the repository being built. In this way, it ensures no loss of type information.

C.3.3 Dependency Configuration

From COLLECTOR's analysis, we conclude that COLLECTOR has one data dependency namely the content of the XML file produced by HARVESTER. However, we do not need to configure this file dependency. The reason for this is the following. By configuring this file dependency, we would have INCRE checking the XML file for modification. However, the incremental strategy of HARVESTER implies COLLECTOR to read the XML file produced by HARVESTER and copying the types omitted by the HARVESTER. This means that COLLECTOR should always read the XML file produced by HARVESTER. This latter implies that checking the XML file for modification is redundant. Hence, we do not need to configure this file dependency for COLLECTOR.

C.3.4 Comparison Configuration

We do not need to configure fields-restrictions for COLLECTOR because COLLECTOR does not have any object dependencies.

C.4 Realization of Incremental LOLA

C.4.1 Analysis

Input: Superimposition selectors and program elements with relations between them.
Output: Sets of selected program elements per superimposition selector.
Processing: LOLA iterates over each superimposition selector and evaluates its prolog query by means of a prolog engine. The evaluation of a prolog query results in the selection of a set of program elements [25].
Dependencies: The following three data types influence LOLA's evaluation of a superimposition selector:

- Syntax of the selector's prolog query. It is clear that changing the syntax of a query may result in a different selection of program elements.
- Type information relevant for the prolog query. Each prolog query selects its program elements based on the existence of its own relevant type information. For instance, the simple prolog query *isClass(C)*, is only interested in classes rather than fields and methods. A change to one of this set of type information may affect the selection of program elements. For instance, adding a class affects the prolog query *isClass(C)* because the added class should be selected.
- Result set of other superimposition selectors. A selector to be evaluated may depend on the result set of another selector B when and only when:
  1. The selector to be evaluated selects its program elements based on the existence of one or more annotations.
  2. At least one of the above annotations is superimposed on the program elements selected by selector B.

  Havinga [25] elaborates on the consequences of having dependencies between superimposition of annotations and selectors.

Motivation Incremental Performance LOLA re-evaluates all predicate selectors in every compilation. This strategy is inefficient as it leads to redundant repeats of evaluations. By applying our incremental rebuilding solution to LOLA, we should minimize these redundant evaluations and improve LOLA's performance.

C.4.2 Alternative Algorithm for Incremental Evaluation

Like the other compilation modules, we can tell which predicate selectors have been evaluated in the previous compilation by checking LOLA's dependent data. But, for LOLA, we cannot use our XML configuration file to accomplish this. The reason for this is the following limitation of the XML configuration file. It does not provide any XML tags yet to configure inter-dependencies between input objects. In other words, we cannot configure LOLA's third dependency, which is the dependency between predicate selectors. Because of this limitation, LOLA forms an exception to our normal procedure for adding incremental performance to a compilation module. To overcome this limitation, we have developed an alternative hard-coded Java pre-process algorithm. This algorithm realizes incremental evaluation of predicate selectors by completing the following six steps:

1. **Collect all user-provided predicate selectors.** The first step is to collect all predicate selectors. The next steps divide these predicate selectors into two lists:

   (a) `toBeSkipped`. This list contains predicate selectors for which we do know for sure that they have been evaluated in the previous compilation.

   (b) `toBeProcessed`. This list contains the remaining predicate selectors, which are to be evaluated by LOLA.

2. **Divide the predicate selectors based on `toBeCheckedByINCRE`.** The predicate selectors are first divided on their boolean attribute `toBeCheckedByINCRE`. This attribute is only true for predicate selectors that are interested in minimal type information. By this, we mean the following. While evaluating a predicate selector, LOLA's prolog engine searches only for relevant type information. E.g., for predicate `isClass(C)`, the prolog engine is only interested in the names of classes rather than methods and parameters. One can imagine that checking this relevant type information for changes could potentially result in a huge overhead. Hence, we have decided to avoid checking large amount of type information by introducing the attribute `toBeCheckedByINCRE`. When this attribute is set to `false` for a specific predicate selector, we do not check its other dependent data and recognize that predicate selector as to be processed by LOLA. Listing C.3 shows Java code that realizes this first dividing of predicate selectors.

```
1  public ArrayList splitSelectors(ArrayList selectors) throws ModuleException
2  {
3    INCRE incre = INCRE.instance();
4    DataStore ds = DataStore.instance();
5
6    // Step 1: whether selector needs to be checked by INCRE
7    // First split is based on the selector's attribute toBeCheckedByINCRE
8    ArrayList toBeProcessed = new ArrayList();
9    ArrayList toBeSkipped = new ArrayList();
10   ArrayList toBeMoved = new ArrayList();
11   Iterator predicateIterStep1 = selectors.iterator();
12   while (predicateIterStep1.hasNext()){
13     PredicateSelector predSel = (PredicateSelector)predicateIterStep1.next();
14     if(predSel.getToBeCheckedByINCRE())
15       toBeSkipped.add(predSel);
16     else toBeProcessed.add(predSel);
17   }
18
19   // split on remaining dependencies...
20
21 } // end method splitSelectors
```

Listing C.3: Step one of dividing algorithm LOLA

The above code iterates over each predicate selector (line 12) and checks their boolean attribute `toBeCheckedByINCRE` (line 14). When this attribute is `true`, the predicate selector is added to the `toBeSkipped` list (line 15). Otherwise, it is added to the `toBeProcessed` list (line 16).

3. **Divide the predicate selectors based on query syntax.** The second dividing of predicate selectors is made based on the syntax of the query. When this syntax has changed since a last compilation, we do not know for sure whether the result of the predicate selector will be the same as in the previous compilation. Hence, we add this selector to the `toBeProcessed` list. Listing C.4 shows Java code that realizes this second dividing of predicate selectors.

```java
1  // Step 2: split further based on syntax of query
2  // When query modified => process selector
3  Iterator predicateIterStep2 = toBeSkipped.iterator();
4  while (predicateIterStep2.hasNext()){
5    PredicateSelector predSel = (PredicateSelector)predicateIterStep2.next();
6    PredicateSelector copySel = (PredicateSelector)incre.findHistoryObject(
         predSel);
7
8    if(copySel!=null){// check query syntax
9      if(!(predSel.getQuery()).equals(copySel.getQuery()))
10       toBeMoved.add(predSel);
11   }
12   else toBeMoved.add(predSel);
13 }
14 // move selectors from toBeSkipped to toBeProcessed
15 moveSelectors(toBeMoved,toBeSkipped,toBeProcessed);
```

Listing C.4: Step two of dividing algorithm LOLA

The above code iterates over each predicate selector still to be processed by LOLA (line 4). For each of these predicate selectors, we check whether the syntax query has changed since a last compilation. We check this by first asking an instance of `INCRE` for an old copy of the predicate selector (line 6). When this copy is not `null`, we compare the syntax of both queries (line 9). When the syntax differs or when the old copy cannot be found, we move the predicate selector from the `toBeSkipped` list to the `toBeProcessed` list. This way, only predicate selectors whose query syntax have not changed, are kept in the `toBeSkipped` list.

4. **Divide the predicate selectors based on type information.** The third dividing of predicate selectors is based on the predicate selector's relevant type information (see step 2 for explanation of relevant type information). Listing C.5 shows Java code that realizes this third dividing of predicate selectors

```java
// Step 3: split based on query specific type information
// When type information modified => process selector
Iterator predicateIterStep3 = toBeSkipped.iterator();
ArrayList currentTYM = incre.getAllModifiedPrimitiveConcerns(ds);
ArrayList historyTYM = incre.getAllModifiedPrimitiveConcerns(incre.history);
Module lola = incre.getConfigManager().getModuleByID("LOLA");
MyComparator comparator = new MyComparator("LOLA");
while (predicateIterStep3.hasNext()){
  PredicateSelector predSel = (PredicateSelector)predicateIterStep3.next();
  lola.addComparableObjects(predSel.getTYMInfo());
  comparator.clearComparisons();
  if(!comparator.compare(currentTYM,historyTYM)) // compare type information
    toBeMoved.add(predSel);
  lola.removeComparableObjects(predSel.getTYMInfo());
}

// move selectors from toBeSkipped to toBeProcessed
```

```
moveSelectors(toBeMoved,toBeSkipped,toBeProcessed);
```

Listing C.5: Step three of dividing algorithm LOLA

The above code iterates over each predicate selector still to be processed by LOLA (line 8). For each of these predicate selectors, we check whether its relevant type information has changed since a last compilation or not. We check this by first setting up an instance of `MyComparator` (line 7). Then we tell this comparator to compare objects solely on the predicate selector's relevant type information by means of its `addComparableObjects` method (line 10). Finally, we compare all concerns from the repository being built and the backup repository to see whether there is a mismatch in the relevant type information or not (line 12). When this is the case, we move the current predicate selector from the `toBeSkipped` list to the `toBeProcessed` list. This way, only predicate selectors whose relevant type information have not changed, are kept in the `toBeSkipped` list.

5. **Divide the predicate selectors based on inter-dependencies.** The fourth dividing of predicate selectors is based on the inter-dependencies between predicate selectors. From LOLA's analysis, we recall that a predicate selector A depends on a predicate selector B when and only when they satisfy the following two conditions:

   (a) Predicate selector A selects its program elements based on the existence of one or more annotations.

   (b) At least one of these annotations is superimposed on the program elements selected by predicate selector B.

   Based on these two conditions, we can retrieve all dependent predicate selectors for a specific predicate selector A. When one of these dependent predicate selectors has changed since a previous compilation, we do not know for sure whether the result of predicate selector A will be the same as in the previous compilation. In other words, when a predicate selector is recognized as to be skipped by LOLA, all its dependent predicate selectors should be recognized as to be skipped as well. If not, then the predicate selector must be processed by LOLA. Listing C.6 shows this fourth and final dividing of predicate selectors.

```
// Step 4: split selectors based on dependent selectors
ArrayList depSelectorsList = new ArrayList();
boolean restart = true;
while(restart){ // enter main loop
  restart = false;
  depSelectorsList.clear();
  if(!toBeSkipped.isEmpty()){
    Iterator predicateIterStep4 = toBeSkipped.iterator();
    while (predicateIterStep4.hasNext()){
      // for each selector gather dependent selectors
      PredicateSelector predSel = (PredicateSelector)predicateIterStep4.next();
      if(!predSel.getAnnotations().isEmpty()){
        Iterator annots = predSel.getAnnotations().iterator();
        while(annots.hasNext()){
          // for each annotation find selectors superimposing it
          String annotToFind = (String)annots.next();
          Iterator annotBindingIter = dataStore.getAllInstancesOf(
              AnnotationBinding.class);
          while (annotBindingIter.hasNext()){
            AnnotationBinding annotBind = (AnnotationBinding) annotBindingIter.
                next();
20             Iterator annotRefs = annotBind.annotationList.iterator();
21             while(annotRefs.hasNext()){
22               ConcernReference annotRef = (ConcernReference)annotRefs.next();
23               Type annotation = (Type)annotRef.getRef().
                   getPlatformRepresentation();
24               if(annotation.getUnitName().equals(annotToFind)){
25                 depSelectorsList.add(annotBind.getSelector().getRef());
26                 continue;
27               }
28             }
29           }
30         }
31       }
32 
33       // check whether dependent selectors are in toBeSkipped
34       // If not, then process current selector and restart this step
35       if(!depSelectorsList.isEmpty()){
36         Iterator depSelectors = depSelectorsList.iterator();
37         while(depSelectors.hasNext()){
38           SelectorDefinition depSelector = (SelectorDefinition)depSelectors.
               next();
39           Iterator selExpressions = depSelector.selExpressionList.iterator();
40           while(selExpressions.hasNext()){
41             SimpleSelExpression simpleSel = (SimpleSelExpression)selExpressions
                 .next();
42             if(simpleSel instanceof PredicateSelector){
43               if(toBeProcessed.contains((PredicateSelector)simpleSel) &&
                   toBeSkipped.contains(predSel)){
44                 moveSelector(predSel,toBeSkipped,toBeProcessed);
45                 restart = true;
46               }
47             }
48           }
49         }
50       }
51 
52       if(restart) break;
53     } // end selector iteration
54   }
55 }// end main loop
```

Listing C.6: Step four of dividing algorithm LOLA

The above code first enters a main loop (line 4). Within this loop, we iterate over each predicate selector still to be processed by LOLA (line 9). For each of these predicate selectors, we check whether the selector selects its program elements based on the existence of any annotation (line 12). If so, we collect these annotations and for each of these annotations, we search for any predicate selector that is used to superimpose that annotation (lines 13–30). Then, we check whether all of these predicate selectors are available in the `toBeSkipped` list. If not, we move the current predicate selector from the `toBeSkipped` list to the `toBeProcessed` list (line 44). Finally, we indicate that there has been a change in these lists (line 45). This loop repeats until nothing changes. This way, only predicate selectors whose dependent selectors are all in the `toBeSkipped` list, are kept in the `toBeSkipped` list.

6. **Call copy operation for `toBeSkipped` predicate selectors.** The final step of this algorithm is calling a copy operation for each predicate selector in the `toBeSkipped` list. Listing C.7

shows code that accomplishes this.

```
1  // Step 5: resolve answers of skipped selectors
2  Iterator predicateIterStep5 = toBeSkipped.iterator();
3  while (predicateIterStep5.hasNext()){
4    PredicateSelector predSel = (PredicateSelector)predicateIterStep5.next();
5    if(!predSel.resolveAnswers()){
6      // answers cannot be resolved, re-evaluate selector
7      toBeMoved.add(predSel);
8    }
9  }
10
11 // move selectors from toBeSkipped to toBeProcessed
12 moveSelectors(toBeMoved,toBeSkipped,toBeProcessed);
13
14 // return the list containing all selectors still to be processed
15 return toBeProcessed;
```

Listing C.7: Step five of dividing algorithm LOLA

First, we iterate over each predicate selector to be skipped by LOLA (line 3). Then, for each of these predicate selectors, we call their copy operation that is called `resolveAnswers` (line 5). This method searches the backup repository for all program elements selected by the predicate selector in the previous compilation. Finally, if these programs elements cannot be resolved, we move the predicate selector to the `toBeProcessed` list (line 7). This security mechanism ensures us that a predicate selector is still evaluated when its copy operation fails.

C.5 Realization of Incremental SECRET

C.5.1 Analysis

Input: Concerns with a concrete filtermodule ordering (calculated by FILTH) and a user-provided filter specification file.

Output: A HTML conflict report showing where and how semantic conflicts occur.

Processing: When multiple filtermodules are imposed on the same joinpoint, certain semantic conflicts may be introduced. SECRET aims to reason about these kinds of semantic conflicts for each concern with one or more filtermodule orderings. It performs a static analysis on the semantics of the filters and detects possible conflicts. The used model is, through the use of an XML input specification, completely user adaptable [16, 52].

Dependencies: The following x data types influence SECRET's processing of a concern:

- Configuration of mode. SECRET has three different modes of operation namely:

  normal SECRET only checks the first filtermodule ordering selected by FILTH.
  redundant SECRET checks all other possible filtermodule orderings.
  progressive SECRET checks all other possible filtermodule orderings and selects the first one without semantic conflicts.

  Changing this mode for a concern may result in a different filtermodule ordering selected and less or more conflicts detected.
- Structural content of XML configuration file. A user can influence SECRET's analysis by adding or removing semantic constraints. While analyzing a filtermodule or-

Table C.1: Data of a filtermodule ordering analyzed by SECRET

Instance of	SECRET data
FilterModuleOrder	order of the contained filtermodule references
FilterModuleReference	field *ref* (contains a FilterModule object)
FilterModule	field *inputfilters* (output filters are not analyzed)
Filter	field *filterelements*
FilterElement	fields *conditionPart* and *matchingPatterns*
ConditionLiteral	qualified name of the referenced condition
Or	fields *left* and *right*
MatchingPart	name of target and selector
SubstitutionPart	name of target and selector
MessageSelector	field *typeList*
DotNETAttribute	fields *value* and *target*
DotNETMethodInfo	field *name*

dering, SECRET detects semantic conflicts for each unsatisfied constraint. Changing these semantic constraints may influence the number of semantic conflict detections.

- All semantic annotations of methods that are called by a meta filter superimposed on the input concern. SECRET uses these semantics to detect semantic conflicts. Changing these semantics may influence the number of semantic conflict detections.
- *SECRET data* of the filtermodule ordering selected by FILTH and all other possible filtermodule orderings calculated by FILTH. *SECRET data* of a filtermodule ordering is data that is used by SECRET while analysing that filtermodule ordering. Table C.1 defines this data. Note that the SECRET data of the filtermodule orderings that are not selected by FILTH are only a dependency when SECRET's mode is either redundant or progressive. In normal mode, these filtermodule orderings are not analyzed.

Motivation Incremental Performance SECRET performs all semantic analyses in every compilation. This strategy is inefficient as it leads to redundant repeats of analyses. By applying our incremental rebuilding solution to SECRET, we should minimize these redundant analyses and improve SECRET's performance.

C.5.2 Copy Operation

Each concern analyzed by SECRET, receives an object called "SECRETReports". This object contains a report of the semantic analyze performed by SECRET. The copy operation of SECRET searches the backup repository for these kinds of objects and adds them to the repository being built.

C.5.3 Dependency Configuration

From SECRET's analysis, we conclude that four data types influence a semantic analysis of a concern. Listing C.8 presents a configuration of these four data dependencies:

```
<dependencies>
  <dependency type="FILE" name="SECRETConfigFile" isAdded="false">
```

```
 3     <path>
 4       <node type="CONFIG" nodevalue="SECRETConfigFile"></node>
 5     </path>
 6   </dependency>
 7   <dependency type="OBJECT" name="SECRETMode" store="true" lookup="true">
 8     <path>
 9       <node type="CONFIG" nodevalue="SECRETMode"></node>
10     </path>
11   </dependency>
12   <dependency type="OBJECT" name="semantics" store="true" lookup="true">
13     <path>
14       <node type="METHOD" nodevalue="Composestar.Core.CKRET.SECRET.
             getSemanticAnnotations"></node>
15     </path>
16   </dependency>
17   <dependency type="OBJECT" name="filtermoduleorders">
18    <path>
19      <node type="DYNAMIC" nodevalue="FilterModuleOrders"></node>
20    </path>
21   </dependency>
22 </dependencies>
```

Listing C.8: Dependency Configuration of SECRET

The first two dependencies are referenced by project configurations. The first dependency is SECRET's configuration file. This file is referenced by a project configuration called "SECRETConfigFile" (line 4). The second dependency is SECRET's mode of operation. This mode is referenced by a project configuration called "SECRETMode" (line 9). SECRET's third configured dependency refers to the semantic annotations of the user-provided base program. These annotations are obtained by a new helper method called "getSemanticAnnotations" (line 14). The fourth configured dependency refers to all possible filtermodule orderings of a concern. These orderings are referenced by a concern's dynamic object called "FilterModuleOrders" (line 19).

C.5.4 Comparison Configuration

SECRET's fourth dependency returns a list of all possible filtermodule orderings for a concern rather than only the SECRET data listed in Table C.1. This means that it returns more data than is necessary. However, just like FILTH's case, we can restrict this data further by means of our comparison configuration. Appendix B presents a configuration of this restriction. This configuration restricts the comparison of instances of `FilterModuleReference` to the data listed in Table C.1.

C.6 Realization of Incremental ILICIT

C.6.1 Analysis

Input: .NET assemblies from disk and CONE's interception specification file.
Output: Weaved .NET assemblies on disk.
Processing: ILICIT is a .NET Intermediate Language (IL) Weaver. It uses CONE's weave specification file to insert (weave) additional IL-code in the .NET assemblies produced by

RECOMA. The resulting weaved assemblies enforce the Compose⋆ runtime to execute the declared aspects.

Dependencies: The following four data types influence ILICIT's weaving of a .NET assembly:

- Content of the input .NET assembly. When the content of a .NET assembly differs before weaving, the content also differs after weaving. This is because ILICIT only inserts rather than deletes and modifies IL-code.
- Application start-up object. Each .NET application has one start-up object. At the entry-point of this start-up object, ILICIT weaves some special IL-code. This IL-code enforces the Compose⋆ runtime to deserialize the repository serialized at the end of compilation. When the start-up object changes, the place of this special IL-code needs to change accordingly.
- Configuration of the debug level of Compose⋆ runtime. ILICIT weaves the value of this configuration at the entry-point of the application. This enforces the level of debugging while executing a Compose⋆ program.
- Content of weave specification file. A weave specification file tells ILICIT what and where to weave in .NET assemblies. The following data types form the weave specification for a specific .NET assembly:
  1. Concerns with one or more filtermodules superimposed and which have at least one method that may be invoked by the specified .NET assembly.
  2. Concerns which instantiations are intercepted by the weaver and which are actually instantiated by the specified .NET assembly. The weaver intercepts the instantiation of the following concerns:
     (a) Concerns that form the implementation part of the composition filters.
     (b) Concerns with one or more filtermodules superimposed.
  3. The following castings performed by the specified .NET assembly:
     (a) Castings to concerns with one or more filtermodule internals superimposed.
     (b) Castings to internals of superimposed filtermodules.
  4. Concerns with one or more outputfilters superimposed and which have at least one method that may be invoked by the specified .NET assembly.

Motivation Incremental Performance ILICIT reweaves all assemblies in every compilation. This strategy is inefficient as it leads to redundant repeats of weavings. By applying our incremental rebuilding solution to ILICIT, we should minimize these redundant weavings and improve ILICIT's weaving performance.

C.6.2 Copy Operation

The copy operation of ILICIT, only needs to add the weaving target of a skipped assembly to its internal list of woven assemblies. This is necessary because the compilation module BACO (Bulk Assembly Copy) relies on this information to work correctly.

C.6.3 Dependency Configuration

From ILICIT's analysis, we conclude that there are seven data types that influence ILICIT's weaving of an assembly. We have added these seven data dependencies to our XML configuration file. Listing C.9 presents this configuration.

```
 1 <dependencies>
 2   <dependency type="FILE" name="assembly" isAdded="false"></dependency>
 3   <dependency type="OBJECT" name="ApplicationInfo" store="true" lookup="true">
 4     <path>
 5       <node type="CONFIG" nodevalue="ApplicationStart"></node>
 6     </path>
 7   </dependency>
 8   <dependency type="OBJECT" name="RunDebugLevel" store="true" lookup="true">
 9     <path>
10       <node type="CONFIG" nodevalue="RunDebugLevel"></node>
11     </path>
12   </dependency>
13   <dependency type="OBJECT" name="ConcernsWithFMO">
14     <path>
15       <node type="METHOD" nodevalue="Composestar.DotNET.ILICIT.ILICIT.
           getConcernsWithFMO"></node>
16     </path>
17   </dependency>
18   <dependency type="OBJECT" name="CastingInterceptions">
19     <path>
20       <node type="METHOD" nodevalue="Composestar.DotNET.ILICIT.ILICIT.
           castingInterceptions"></node>
21     </path>
22   </dependency>
23   <dependency type="OBJECT" name="InstantiationClasses">
24     <path>
25       <node type="METHOD" nodevalue="Composestar.DotNET.ILICIT.ILICIT.
           getAfterInstantiationClasses"></node>
26     </path>
27   </dependency>
28   <dependency type="OBJECT" name="ConcernsWithOutputFilters">
29     <path>
30       <node type="METHOD" nodevalue="Composestar.DotNET.ILICIT.ILICIT.
           getConcernsWithOutputFilters"></node>
31    </path>
32   </dependency>
33 </dependencies>
```

Listing C.9: Dependency Configuration of ILICIT

The first dependency returns the input object itself which is the filename of an assembly to be weaved. This dependency is of type FILE and is called "assembly" (line 2). The second dependency refers to the start-up object of the user-provided project. This object is referenced by the project configuration called "ApplicationStart" (line 5). ILICIT's third configured dependency is the level of debugging output at runtime. This level is referenced by the project configuration called "RunDebugLevel" (line 10). The last four dependencies return the four data types that form the weave specification file for an assembly (see Section C.6.1 for a description of these four data types). Each of these four data types is obtained by a new helper method (lines 13-32).

C.6.4 Comparison Configuration

ILICIT has six configured object dependencies. Each of the data dependencies refer to either a primitive or a list of primitives. These primitives cannot further be restricted to optimize repository entity comparison. Hence, there are no field-restrictions configured for ILICIT.