Profiling of SCOOP Programs

Master Thesis

Martino Trosi
ETH Zürich
trosim@student.ethz.ch

November 1st, 2009 - April 30th, 2010

Supervised by:
Benjamin Morandi
Prof. Dr. Bertrand Meyer
Abstract

SCOOP (Simple Concurrent Object-Oriented Programming) [18] is a model and practical framework for building concurrent applications. It comes as a refinement of the Eiffel [15] programming language and is in the process of being integrated into the research version of EiffelStudio, called EVE [16].

Many examples have proven SCOOP’s simplicity, but some of these show bad performance. Although the user-friendliness of SCOOP eases the development of concurrent programs, there is no systematic way to measure its performance yet.

We developed a SCOOP profiler to help users understand why a particular SCOOP program is running slowly, discover the bottlenecks, and modify the code to improve the run time. Moreover, the profiler would show us eventual performance problems in the SCOOP model and implementation. We prove it useful by profiling a couple of example programs, reporting on our process and results.
Acknowledgments

I would like to thank very much my mentor, Prof. Bertrand Meyer, for letting me work in his group and for the constructive critics during this thesis.

Many thanks to Benjamin Morandi for the great work on the SCOOP compiler and for the support he provided me while developing the profiler, not to mention the exhausting review of this report.

I also wish to thank Sebastian Nanz, Damien Müllhaupt and the whole group working on SCOOP for many valuable suggestions and helpful comments. It was a pleasure to work with all of you!

Thanks to my girlfriend Laura and my family for not going mad while I was working with baboons and barbers, trying to explain what processors and threads are and how the SCOOP profiler works. Your support during the whole course of my studies has been very significant.
# Contents

1 Introduction 6
   1.1 Motivation ............................................. 6
   1.2 Goals .................................................. 6
   1.3 Research Contribution ................................. 7
   1.4 Outline ................................................. 7

2 Related Work 8
   2.1 SCOOP .................................................. 8
   2.2 Methods and Metrics .................................. 9
   2.3 Object-Oriented Proposals for Instrumentation ....... 12
   2.4 Problem of current solutions ......................... 12

3 Design 13
   3.1 Metrics ................................................. 13
   3.2 Analysis Type ......................................... 14
   3.3 Representation of Profile Data ....................... 15
   3.4 Format of Profile Data ................................ 17
   3.5 Profiling Hooks ...................................... 18
   3.6 User Interface ....................................... 20

4 Implementation 22
   4.1 Profiling Information ................................ 22
   4.2 Events .................................................. 23
   4.3 Writing Profile Data .................................. 25
   4.4 Processing Profile Data ............................... 26
   4.5 Abstraction ........................................... 27
   4.6 Loader and Visitor Interaction ....................... 28
   4.7 Visualization ......................................... 30

5 Evaluation 35
   5.1 Demonstration Usage .................................. 35
   5.2 Observer Effect ....................................... 42
   5.3 Problems .............................................. 43

6 Developers Guide 45
   6.1 Configuration .......................................... 45
   6.2 Profile Information ................................... 46
   6.3 Events .................................................. 46
# Table of Contents

6.4 Profile Abstractions ........................................... 46  
6.5 Profiler Wizard .................................................. 47  

7 User Guide ......................................................... 48  
  7.1 Enabling the Profiler ........................................... 48  
  7.2 Running the wizard ........................................... 49  
  7.3 Visualizing Results .......................................... 51  

8 Future Work ....................................................... 53  
  8.1 Static Metrics ................................................ 53  
  8.2 Dynamic Metrics ............................................ 53  
  8.3 Events .......................................................... 54  
  8.4 Data Serialization .......................................... 54  
  8.5 Graphical Wizard ............................................ 55  
  8.6 Profile Analysis .............................................. 55  

9 Conclusions ....................................................... 56  
  9.1 Critique ....................................................... 56  
  9.2 Personal Remarks ........................................... 57  

A Listings .......................................................... 58  
  A.1 Gathering of profile information ......................... 58  
  A.2 Collection of events ....................................... 59  
  A.3 Loaders and Visitors ....................................... 62  
  A.4 Examples ..................................................... 66  

B Glossary .......................................................... 68
Chapter 1

Introduction

In the following sections we motivate this project, discuss the problems with current solutions and, at the end of the chapter, summarize the outline of the thesis.

1.1 Motivation

SCOOP (Simple Concurrent Object-Oriented Programming) [18] is a model and practical framework for building concurrent applications. It comes as a refinement of the Eiffel [15] programming language and is in the process of being integrated into the research version of EiffelStudio, called EVE [16].

Many examples have proven SCOOP’s simplicity, but some of these show bad performance. Although the user-friendliness of SCOOP eases the development of concurrent programs, there is no systematic way to measure its performance yet.

We developed a SCOOP profiler to help users understand why a particular SCOOP program is running slowly, discover the bottlenecks, and modify the code to improve the run time. Moreover, the profiler would show us eventual performance problems in the SCOOP model and implementation. We prove it useful by profiling a couple of example programs, reporting on our process and results.

1.2 Goals

The objectives of this thesis are:

- Design a profiler framework, and visualization tools.
- Clearly document the design choices and metrics.
- Implement and integrate the profile generation and wizard into EVE.
• Properly document and test all the code.
• Profile a few example applications and their improved versions in terms of performance.
• File eventual bugs in the SCOOP compiler and library.

1.3 Research Contribution

The main contribution of this thesis is twofold. First, using the SCOOP profiler, the user can have a clear idea of what happens during the execution of a program and find ways to improve the performance by means of a clear visualization.

Second, by profiling worst-case programs and ad-hoc hedge cases, the people who work on the implementation of the SCOOP model and library can evaluate the strength of the implementation and improve it. The SCOOP profiler can also be used to compare the model and its performances with traditional semaphore based approaches [2].

1.4 Outline

General approaches to profiling and work related to this thesis are introduced in Chapter 2. This chapter presents the fundamentals on which our project is based, and sets the starting point for many design decisions.

The design of the SCOOP profiler is explained in Chapter 3. Some of the main architectural choices are described in detail.

Chapter 4 focuses on the implementation of the SCOOP profiler, giving insight in the technical details from events collection through visualization.

Developers are provided with an extensive guide to understand and edit the solution presented in this thesis in Chapter 6. Parts of code strictly related to future development of the profiler are shown in detail.

A small and straightforward user guide is presented in Chapter 7. This chapter goes through a step-by-step usage of the profiler and supports the user with instructions and screenshots.

In Chapter 8 we describe future work that can be done to improve the profiler and its usability.

Chapter 9 concludes the main issues of this thesis and summarizes what we have learned from this project.
Chapter 2

Related Work

This chapter starts by explaining how the current implementation of the SCOOP compiler works, so to clarify the base on which many design decisions have been taken. In the subsequent sections actual solutions and ideas in the field are discussed.

2.1 SCOOP

A lot of work has been done on SCOOP in the last years. We try to summarize the most important steps in the development of SCOOP and its way towards an integration in EVE. In the following sections we will review briefly the concepts behind the SCOOP compiler and the SCOOP library.

Historical Notes

The SCOOP concept was first introduced informally by Bertrand Meyer in the early 90s. The model found its way into a couple of articles [8] before being described in the book *Object Oriented Software Construction* [9], published in 1997.

Piotr Nienaltowski has completely reworked the model, in his dissertation [11] published 2007, under supervision of Bertrand Meyer. The SCOOP library has been implemented and the first version of SCOOP compiler was build, based on the GOBO [17] Eiffel Compiler.

The first efforts to reimplement and extend the compiler into EVE have been done by Patrick Huber during his master thesis [6], supervised by Benjamin Morandi and Bertrand Meyer. Current state of SCOOP is described in [10].
Listing 2.1: *Eiffel*: Example compilation: original code

```
--- In class BABOO (original source).
announce (a_rope: separate ROPE)
--- Announce.
    do
        io.put_string (out + " announcing %N")
    a_rope.announce (Current)
end
```

Compiler

For each processed class, the SCOOP compiler produces two new classes: these are called client and proxy classes. Proxies ensure that the implementation classes (clients) are accessed following the rules of the SCOOP model. The mechanism uses, as the name says, the proxy pattern and implies a chain of local calls to achieve a single SCOOP call in the original source file.

The proxy and client classes make use of the SCOOP library as a centralized scheduler to treat the requests of routines that need to lock arguments and ensure the correct behavior of wait conditions.

Thus, the calls that happen in the generated code do not reflect anymore the logical call to a feature, as intended by the programmer. Let us compare, as an example, the original text written by the user in Listing 2.1, and the code generated by the SCOOP compiler: Listing 2.2 shows the code in the proxy class and Listing 2.3 the one generated in the client class.

The generated proxy class contains calls to `scoop_asynchronous_execute` and `scoop_synchronous_execute`. Synchronous calls are due to locking of arguments, lock passing, and wait-by-necessity, every other call can be asynchronous.

Figure 2.1 illustrates the three different types of SCOOP calls. The first two are a) an asynchronous call, and b) a synchronous call without separate attached arguments, and they are not very interesting from the SCOOP point of view because they do not need any synchronization mechanism: no locking happens and thus it is not possible to have wait conditions. Of course the target of these calls has to be already locked. The third call c) is more interesting: it has separate arguments, handled by the scheduler.

2.2 Methods and Metrics

Sahni and others [12] analyze general metrics for parallel programs, such as speedup, efficiency, and scalability. The paper discusses different variations of these metrics and points out similarity and differences. Furthermore, they propose a model for parallel computers.

Denda [1], in his diploma thesis, proposes a prototype profiler for a C based threading library. In the first part of the paper he classifies metrics, discussing differences between time and count measurements, exact versus statistical met-
Listing 2.2: Eiffel: Example compilation: proxy code

```eiffel
-- In class SCOOP SEPARATE BABOON (proxy class, produced by the SCOOP compiler).
announce (a_caller_: SCOOP SEPARATE_TYPE; a_rope: SCOOP SEPARATE ROPE)

local
aux_scoop_a_rope: SCOOP SEPARATE ROPE
scoop_passing_locks: BOOLEAN
scoop_locked_processors_stack_size,
  scoop_synchronous_processors_stack_size: INTEGER

if a_rope /= void and then (a_rope.processor_ /= void) then
aux_scoop_a_rope := a_rope
if aux_scoop_a_rope.processor_ = void then aux_scoop_a_rope.
  set_processor_ (a_caller_.processor_) end
if a_caller_.processor_.locked_processors_has (  
aux_scoop_a_rope.processor_) then scoop_passing_locks :=
  True end

if scoop_passing_locks then
scoop.locked_processors_stack_size := processor_.
  locked_processors_count
scoop.synchronous_processors_stack_size := processor_.
  synchronous_processors_count
processor_.locked_processors.push_whole_stack (a_caller_.
  processor_.locked_processors)
processor_.synchronous_processors.push_whole_stack (a_caller_.
  processor_.synchronous_processors)
scoop.synchronous_execute (a_caller_, agent implementation_.
  announce(aux_scoop_a_rope))
processor_.synchronous_processors.trim (  
scoop.synchronous_processors_stack_size)
processor_.locked_processors.trim (  
scoop.locked_processors_stack_size)

else
scoop.asynchronous_execute (a_caller_, agent implementation_.
  announce(aux_scoop_a_rope))

end
```
Listing 2.3: Eiffel: Example compilation: client code

```
1  -- In class BABoON (client class, produced by the SCOOP compiler).
2  announce (a_rope:SCOOP SEPARATE _ ROPE)
3  do
4      invariant_disabled := True
5  separate_execute_routine ([a_rope . processor_], agent
6      announce_scoop_separate_baboon_enclosing_routine (a_rope),
7      Void, Void)
8  invar iant_disabled := False
9  end
10 announce_scoop_separate_baboon_enclosing_routine (a_rope: SCOOP_SEPARATE _ ROPE)
11    -- Wrapper for enclosing routine ‘announce’.
12    -- Announce.
13    do
14      io . put_string (out + " announcing %N")
15      a_rope . announce (Current, Current)
16  end
17 announce_scoop_separate_baboon_wait_condition (a_rope: SCOOP_SEPARATE _ ROPE): BOOLEAN
18    -- Wrapper for wait-condition of enclosing routine ‘announce’.
19    do
20      Result := True
21  end
```

(a) Asynchronous call (b) Synchronous call (c) Synchronous call, with separate arguments

Figure 2.1: Call flows
Related Work - Object-Oriented Proposals for Instrumentation

metrics, and the operating level of the metric (programming language, operating system, hardware). Monitoring methods for different metrics and instrumentation insertion are discussed as well.

A comparison between metrics for large-scale parallel applications is done in [5]. A technique is described that permits direct quantitative comparison of the guidance provided on real programs by the different metrics considered in the paper.

2.3 Object-Oriented Proposals for Instrumentation

The decorator pattern can be very helpful to non-intrusively profile an object-oriented program, as shown by Duffy and others in [3]. They go into detail on how to apply the pattern to maintain correctness and the external behavior of the original program. The paper discusses where to do invariant validation and makes specific reference to Design-by-Contract [7].

A simple idea called twin-class hierarchy approach is proposed by Factor and others in a paper [4]. It uses two sets of base classes, such as the system library. The first one is composed of the original classes and the second contains their instrumented version. The main issue with this approach is that the instrumented system classes often make use of other system classes to produce the profile data. The paper discusses how to avoid using instrumented system classes in the instrumentation, otherwise we would have sort of an ”instrumentation of the instrumentation”.

2.4 Problem of current solutions

None of the current solutions for implementing profilers described in Chapter 2 is directly usable with programs written with SCOOP. We can bring this back to the design of the actual SCOOP compilation process, explained in Section 2.1.

The main reason is that a separate call, as written in the code, is translated into multiple calls and agents during the SCOOP compilation degree. It is thus necessary to collect data during more than one local call in order to reconstruct the logical call of a separate feature.

For these reasons we could not just use an existing proposal and we had to develop a new solution tailored to the architecture of the current SCOOP model implementation.
Chapter 3

Design

In the following sections we will describe the fundamental design choices of the SCOOP profiler.

3.1 Metrics

We present the metrics we use to profile a program and discuss why we feel they are important and why they may help understand the program behavior.

The most important metric that we want to measure are time spans, in particular the duration of features. Following the SCOOP model we define three time spans during a feature call/application:

- Time until a feature is at the beginning of the action queue. This is the interval from when a new feature call is added to a processor’s queue until the processor has finished executing older calls and is ready to handle the new call. We call this queue time.
- Time until a feature can effectively be executed: during this time span locks on arguments are acquired the processor holds until wait conditions are satisfied. We call this synchronization time.
- Execution time of a feature. This is the normal execution time of a feature, intended as the processing of its body.

Being in a concurrent setting, we are interested in the speedup with respect to a sequential program. These are the metrics that would enable the observation of processors:

- Number of processors in the system
- Time a processor spends waiting/working over time
The first metric gives an idea of the number of processors in the system, in other words it represents how much the program can be parallelized. The second metric can be measured by looking at the SCOOP processors state over the time: either working (executing a routine) or waiting (waiting for other processors). We had two possibilities to interpret these states:

- Thread time: a processor is executing when the respective thread is running, waiting otherwise.
- Execution time: a processor is executing if it is executing the body of a routine, waiting otherwise (acquiring locks, processing wait conditions).

We decided to use the second alternative, mainly because it enables us to distinguish further between waiting, idle, and profiling states, as the states are derived from the generated profile data. This second option is also easier to implement as we do not have to take threads into account and does scale to the distributed setting, where a SCOOP processor is not identifiable with a thread anymore.

In addition, we present two metrics based on counters:

- Number of wait conditions tries before calling each feature: this metric can give an idea of the behavior of multiple features that try to lock the same objects. It might be useful also for evaluating how the scheduler manages the locking requests and wait conditions trials.
- Number of calls per separate feature: this counter basically helps to find out what features represent the hot spots of the program. It is also used to provide more time related information, for example average and standard deviation.

### 3.2 Analysis Type

We want our profile analysis to be decoupled from the profile generation. We have chosen this approach to influence as little as possible the execution of the program and to be able to execute the analysis on a possibly different machine and environment. We took great care in designing a profiler that can evolve with the progress of SCOOP, getting ready for the profiling of a distributed concurrent application that spans over multiple locations (processes, machines, sites).

Besides the fact that SCOOP will eventually be deployed on a distributed environment, it is possible to have long running applications such as servers or mobile applications that often do not offer the facilities to run online analysis tools and graphical wizards.

As a consequence, we decided not to use the EVE internal representation of an Eiffel project to gain information about the profiled program, because this information can possibly not be at our disposal. Instead, we conduct a small analysis at profile generation in order to collect all relevant program information such as the mapping of feature identifiers to feature names for a later analysis.
3.3 Representation of Profile Data

In this section we will discuss the representation of profile data.

The architectures that support Eiffel are very diverse, so a state-copy approach does not seem to be a viable solution. To maintain a high flexibility we decided to use events for the representation of our profile data. They are easily generated and have a wide range of possible representation formats and storages (database, files, streams).

This choice does not constrain us to a post-mortem analysis, leaving open the possibility to implement an online analyzer and on-the-fly visualization. Such a result is outside the scope of this project.

Information

The events have to be self-contained, without the need of external knowledge to interpret them. All recorded times have to be absolute and should be as precise as possible, moreover they should not be approximated until really needed. IDs that depend on a particular compilation or run of the program should be converted to generally valid names. This should be done directly during event generation.

Processors identification is architecture dependent and thus processors cannot be given absolute names. Furthermore processors exist only for the duration of an execution and can generally not be related between different executions. For the actual SCOOP implementation, where processors are implemented with threads, the thread ID is used as the processor identification.

Measurement Points

Figure 3.1 depicts the measurement points, where the events get collected, and the related time intervals. It represents a separate call from processor B to processor A, followed by a profiling action of processor A.

The instants labeled with capital letters represent the collection of some events. In the following we describe the generated events:

- A1: represents the time when a processor is created. Processor A records a \texttt{start} event (processor B records a similar event).
- A2: represents the time when a processor is removed. Processor A records a \texttt{stop} event.
- F1: represents the time of a call, in this case an external call. Processor A records a \texttt{call} event whereas processor B records an \texttt{external\_call} event. The external call event on processor B enables us to reconstruct the original call tree, synchronizing the flows of events of both processors.
- F2: represents the time when the request is at the beginning of the processor queue. Processor A records a \texttt{wait} event.
Figure 3.1: Measurement points

- F3: represents the time when the scheduler executes a wait condition trial, wait conditions may be tested multiple times. Processor A records a `wait_condition` event.

- F4: represents the time when all locks are acquired and all wait condition clauses are true. Processor A records an `application` event.

- F5: represents the time when the feature returns. Processor A records a `return` event.

- P1: represents the beginning of a profiling action, the processor stops executing to write the events to disk. Processor A records a `profiling_start` event.

- P2: represents the end of a profiling action, when the processor finishes writing to disk and resumes normal operation. Processor A records a `profiling_stop` event.

**Time Intervals**

To clarify the meaning of each time interval, represented with lowercase letters, we provide a description:

- i1: represents the interval between the call time and the time when the processor is ready to execute the request, called queue time. During this time the processor A might execute other features, so the queue time, for synchronous features, accounts on the caller processor waiting time.

- i2: represents the interval from when the processor is ready to execute the request, until the scheduler effectively grants execution rights to the request. During this time the scheduler makes sure that all locks on arguments are acquired and that wait conditions are satisfied.

- i3: represents the execution time of the body of a feature. This is not different from sequential profilers.
- i4: represents the total time waiting for a synchronous call to complete. This interval is empty for asynchronous calls.

- i5: represents the duration of a profiling action. These intervals show the influence of the profiler on the execution of a program.

- i6: represents the life time of a processor.

### 3.4 Format of Profile Data

In the following subsections we discuss the different file formats that we have assessed to find the best representation for the profile data. Table 3.1 summarizes the advantages and drawbacks of each approach.

<table>
<thead>
<tr>
<th>Format</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat File</td>
<td>- easy to write</td>
<td>- need to build parser</td>
</tr>
<tr>
<td></td>
<td>- human readable</td>
<td>- need to round times</td>
</tr>
<tr>
<td>Semistructured</td>
<td>- easy to write</td>
<td>- additional dependency</td>
</tr>
<tr>
<td></td>
<td>- human readable</td>
<td>- need to build parser</td>
</tr>
<tr>
<td></td>
<td>- very well supported</td>
<td>- need to round times</td>
</tr>
<tr>
<td>Serialization</td>
<td>- included in base library</td>
<td>- not human readable</td>
</tr>
<tr>
<td></td>
<td>- easy to write and read</td>
<td>- generated size</td>
</tr>
<tr>
<td></td>
<td>- no need to round times</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of file formats

#### Flat File

The first format that comes to mind when saving non-binary data to file is plain text. This is also the format used for the profinfo file, which is generated by the traditional Eiffel profiler. It was immediately clear that the existing format would not fit our need, failing to take into account aspects proper of the SCOOP model, like for example locked processors and caller processor.

Building a custom parser for reading complex text files would mean wasting a lot of development time and introduce issues with data representation, as for example rounding of time values.

#### Semistructured Data

A semi-structured representation is half way between a plain text and a structured format. XML would have been the right choice for saving the data, as we would have had the possibility to create our own schema.

Despite its suitability, we did not choose XML because of the issues that would have been raised in including an XML parser or serialization library in every project.
Serialization

We come to the less versatile format, that is serializing the objects in a proprietary and binary format. This comes in handy as we do not have to worry neither about how to write data to file and read it back nor about how to convert and approximate the information (for example times). The serialization facilities of Eiffel are included in the base library, so we can count on the fact that they will be ready to use in every Eiffel program.

### 3.5 Profiling Hooks

There are many places where to put the profiling hooks. Following the analysis depicted in Table 3.2 and the pros and cons described in the following subsections, we decided to implement the hooks in the SCOOP library.

**C Code**

Starting from the traditional Eiffel profiler, we evaluated the possibility to add the hooks to the C code. This low-level approach would have the advantage of being very optimizable and thus interfere less with the program execution.

The difficulty in adding the profiling instrumentation to the C code is to find out where exactly to place it. Moreover, this would need changes in the code generation step of EiffelStudio, which we would rather prefer to leave untouched.

**Generated Code**

The group working on SCOOP is currently investing in rebuilding the compilation process, thus we considered the possibility to add the profiling code insertion to this process. During the proxy and client classes generation, we could easily throw profiling events from the right place in the code.

The disadvantage of this method is that it requires the extension of the classes used by DEGREE_SCOOP and as these classes were being rewritten we decided not to add to the complexity of that hard work.

**SCOOP Library**

Our discussion lead to considering the option to add the profiling code directly to the SCOOP library. The advantage of this solution is that the code is very clear and non-intrusive and it can be developed further without having complete knowledge of the compilation process.

The drawback of this approach is that we have no access to the feature names from the executing program. It appears clear that we need some mechanism of transferring this information from the compile phase to the execution phase. The information that need to be present at execution time is:
<table>
<thead>
<tr>
<th>Metric</th>
<th>SCOOP Library</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time until at the beginning of the action queue</td>
<td>wrappers in class <code>SCOOP_PROCESSOR</code></td>
<td>not possible, because the generated code does not know about the locking and wait condition trial phase</td>
</tr>
<tr>
<td>Time until execution</td>
<td>wrappers when executing routine request in class <code>SCOOP_SCHEDULER</code></td>
<td>wrappers in generated client/proxy classes</td>
</tr>
<tr>
<td>Execution time</td>
<td>wrappers before and after the call of implementation</td>
<td>wrappers in the code (this could be done by the user, even without the profiling tool)</td>
</tr>
<tr>
<td>Time processing, time waiting</td>
<td>wrappers before and after the call (sum of execution times for the processor)</td>
<td>not possible, because the generated code does not know about the locking and wait condition trial phase</td>
</tr>
<tr>
<td>Processor creation</td>
<td>call when creating a new processor in class <code>SCOOP_SCHEDULER</code></td>
<td>whenever a new object with a new processor is created</td>
</tr>
<tr>
<td>Processor termination</td>
<td>call when removing a processor in class <code>SCOOP_SCHEDULER</code></td>
<td>not possible yet, because the generated code needs to know the number of objects handled by the processors</td>
</tr>
<tr>
<td>Number of objects per processor</td>
<td>not possible yet, each processor should keep a counter for the handled objects and produce the related events</td>
<td>keep counter per processor (need to make each object <code>DISPOSABLE</code>)</td>
</tr>
<tr>
<td>Number of wait condition tries</td>
<td>call when trying wait conditions in class <code>SCOOP_ROUTINE_REQUEST</code></td>
<td>call from the generated wait conditions features</td>
</tr>
</tbody>
</table>

Table 3.2: Comparison of profiling hooks placement
• Class and feature IDs and names: needed to match the IDs accessible at run time to know which feature is being profiled and report readable information in the profile data.

• Arguments’ types: let the profiler know whether a feature is going to lock other objects and thus be executed by the SCOOP_SCHEDULER or whether it is called directly by the SCOOP_PROCESSOR. This information is important to decide which actor, the scheduler or the processor, has to generate the profile events.

The first idea to get by this problem was to generate a mapping class during the SCOOP compilation. This solution is not usable, because the generation of the mapping class will be done after the compilation of the generated code. This would then require a recompilation of the project which, in turn, will regenerate a new mapping class and so on.

We solved this issue by creating an abstraction for profile information that is generated during the DEGREE_SCOOP execution and serialized in the same format as described in Section 3.4. This information is then read at runtime by the profiled program.

The profiling information is not needed if the profiling code is added to the classes generated by the SCOOP compiler, as it would be part of the code.

3.6 User Interface

We describe the overall aspects of the graphical user interface that we foresee for the profiler wizard.

Interaction

The interaction between the user and the profiler should be as simple as possible, especially during the configuration phase.

No library should be added manually to use the profiler. All the needed dependencies should be set up automatically by the compiler, if it finds the separate keyword in the code and the user has activated the profiler in the ECF. EVE should take care of preparing the profile information when the project is compiled. EVE should also automatically record the profile data when the program is run.

To reduce the changes from the traditional profiler, the same wizard structure has to be used and the steps to get to the visualization window have to be modified very little. If someone is used to the profiler wizard, he or she should feel at home when running the SCOOP version of the profiler wizard.
Visualization

The profiler wizard should, on one hand, visualize the processors’ activity over time. Feature calls/applications as well as wait condition trials shall be graphically represented and enable access to detailed information. Moreover, the dependency between processors (lock requests) should be visualized in a manner easy to understand.

On the other hand, general information about the profiled program should be summarized in a schematic way. The summary shall contain overall time measurements and statistical information about them (average, standard deviation). It should be possible to derive the data relative to a particular feature call/application.

An effort should be made to analyze the profile data and represent possible hot spots and problems in a graphical way.
Chapter 4

Implementation

In this chapter we will look at the technical details of the SCOOP profiler implementation.

4.1 Profiling Information

The code for gathering class and feature information is shown in Listing A.1 and A.2 in the appendix, respectively.

Features are considered to be separate if they take at least one attached separate argument; all other features are marked as non-separate. This information is useful when generating the profile data, as feature with separate attached arguments are executed by the SCOOP_SCHEDULER whereas other features (without arguments or with detachable separate arguments) are executed directly by the handling SCOOP_PROCESSOR.

In addition, the size of the events buffer of the collector is saved in the profiler information. Other information can be added, changing one of the following classes:

- **SCOOP_PROFILER_INFORMATION**: contains reference to the directory where to generate profile data, the size of the profile collector buffer, and references to class information objects
- **SCOOP_PROFILER_CLASS_INFORMATION**: contains per-class information, the original class name and id, and the feature information objects
- **SCOOP_PROFILER_FEATURE_INFORMATION**: contains per-feature information, the original feature name and id, and indication about whether it has separate attached arguments
4.2 Events

In the following subsections the event model of the SCOOP profiler is presented. Details about the generation of the events and where it happens in the code are given.

Model

Following is the description of the event classes and the information that they carry. The model is illustrated in the BON diagram in Figure 4.1.

- **SCOOP_PROFILER_EVENT**: base class for all events. It contains just the timestamp of the event.
- **SCOOP_PROFILER_PROCESSOR_EVENT**: base class for processor related events. It adds information about the processor.
- **SCOOP_PROFILER_PROCESSOR_START_EVENT**: this class represents the creation of a processor. It is coupled with **SCOOP_PROFILER_PROCESSOR_END_EVENT**, which represents a termination event. They do not carry other information.
- **SCOOP_PROFILER_FEATURE_EVENT**: base class for feature related events. It provides feature and class IDs and names.
- **SCOOP_PROFILER_FEATURE_EXTERNAL_CALL_EVENT**: represents an external call (a separate call to another processor). It provides the called processor ID.
- **SCOOP_PROFILER_FEATURE_CALL_EVENT**: represents a call. It provides the caller processor ID.
- **SCOOP_PROFILER_FEATURE_WAIT_EVENT**: represents beginning of locking and wait condition trials phase. It contains the IDs of the processors handling the feature arguments.
- **SCOOP_PROFILER_FEATURE_WAIT_CONDITION_EVENT**: represents a trial to satisfy the wait conditions.
- **SCOOP_PROFILER_FEATURE_APPLICATION_EVENT**: represents the beginning of the feature body execution.
- **SCOOP_PROFILER_FEATURE_RETURN_EVENT**: represents the completion of the feature execution.
- **SCOOP_PROFILER_PROFILING_EVENT**: base class for profiling related events. It does not carry other information, as does the event representing the start of a profiling action **SCOOP_PROFILER_PROFILING_START_EVENT** and, similarly, the end of a profiling action **SCOOP_PROFILER_PROFILING_END_EVENT**.
Generation

Figure 4.2 represents the same diagram that is found in Figure 2.1, with the addition of the colored time intervals that are described in Section 3.3 and shown in Figure 3.1. This figure helps us understand where to put the code for collecting events. The locations for feature events are summarized in Table 4.1.

Events representing a call are always generated in the callee processor. An event representing an external call, represented by a red triangle in the figure, is generated in the calling processor, so that the exact call stack can be reconstructed correctly. For the other feature events we have to distinguish between features with or without separate attached arguments.

If a feature has separate attached arguments, it will involve the scheduler, to ensure that the separate arguments’ processors get locked and the wait condition
is satisfied. Feature without separate attached arguments can safely be executed by the processor handling the target of the feature call, thus collecting directly the events; wait and application events happen simultaneously in this case. Listing A.3 shows the collection of events in the scheduler and Listing A.4 shows the collection of events in the proxy.

Wait condition trials events are always generated by the scheduler. Note that features without separate attached arguments do not have wait conditions, all preconditions are evaluated locally and their violation will cause an exception. Listing A.5 shows the collection of wait condition events for features with separate attached arguments.

<table>
<thead>
<tr>
<th>Event</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>Calls are always collected by the callee processor, as shown in Listing A.4.</td>
</tr>
<tr>
<td>Wait</td>
<td>Collected by the scheduler, as shown in Listing A.3.</td>
</tr>
<tr>
<td>Application Return</td>
<td>Collected by the callee processor, as shown in Listing A.4.</td>
</tr>
<tr>
<td>Wait condition trial</td>
<td>Collected by the scheduler, see Listing A.5.</td>
</tr>
</tbody>
</table>

Table 4.1: Feature events collection places.

4.3 Writing Profile Data

When the collector buffer is full, each processor flushes the events that it has generated to disk. The buffer size is controlled via the ECF file. A default value is hardcoded in class `SCOOP_LIBRARY_CONSTANTS`, in case the ECF does not specify a value.

The file names follow this scheme: `<start><end><id><num>.pfi`, where the information in angle brackets have the meaning and format described below:

- `start`: time of the first event contained in the file, number of seconds from the start of the UNIX epoch.
- `stop`: time of the last event contained in the file, number of seconds from the start of the UNIX epoch.
- `id`: integer ID of the processor.
- `num`: file number for the current processor, needed to reorder the files in case there are more than one with the same timestamps.

The class `SCOOP_PROFILER_FILE` models a profile file and it allows access to the information described above.
4.4 Processing Profile Data

Loaders read events from files and represent them as a traversable data structure, they all inherit from the base class \texttt{SCOOP_PROFILER_LOADER}.

The class \texttt{SCOOP_PROFILER_DEFAULT_LOADER} reads all the files present in a directory, using \texttt{SCOOP_PROFILER_FILES_DESERIALIZER}, and builds a queue for each processor. Events for the same processor are ordered by time, files are only read on demand when their content is needed. The queues can be controlled with the features \texttt{continue}, \texttt{delay}, and \texttt{resume}, explained in detail in Section 4.6. All of them take as argument the ID of the processor whose queue has to continue to load events, to delay the next event or to resume loading events, respectively.

Now that we have a mean of creating a stream of events, we need a way to process them, so we let all event classes implement the visitor pattern. We have built some specific visitors to ease the post processing of the profile data. The base class for every visitor is \texttt{SCOOP_PROFILER_EVENT_VISITOR}.

- \texttt{SCOOP_PROFILER_PRINTER_EVENT_VISITOR}: can be used during the development of the loaders or the profile abstractions, it simply displays all the events it visits.

- \texttt{SCOOP_PROFILER_DEFAULT_EVENT_VISITOR}: builds the standard profile abstraction described in the following section. It has to be used with a complete set of data, representing the whole execution of a program.

- \texttt{SCOOP_PROFILER_PARTIAL_EVENT_VISITOR}: adds the ability to build the profile abstraction without having a complete set of data. This is used to profile an arbitrary interval. Time spans that start before the beginning of the interval are not shown, whereas time spans that extends after the interval are displayed as open ended (the time spans are bounded by the interval and marked as having incomplete data).

The model of visitors and loaders is shown on Figure 4.3. Listings \texttt{A.7} and \texttt{A.8} show the base classes for loaders and visitors, respectively.

\textbf{Figure 4.3: BON diagram of visitors and loaders}
4.5 Abstraction

The visitors are used to build an abstraction for fast access to the data. The following list describes the classes that constitute the base representation of a profiled program. The model is shown in Figure 4.4.

- **SCOOP_PROFILER_APPLICATION_PROFILE**: represents the profile of an application. It is used as a factory to create all the other profile objects and provides access to them.

- **SCOOP_PROFILER_PROCESSOR_PROFILE**: represents the profile of a processor. Contains all the calls and profiling actions executed by the processor.

- **SCOOP_PROFILER_CLASS_PROFILE**: represents the profile of a class, providing access to feature profiles.

- **SCOOP_PROFILER_FEATURE_PROFILE**: represents a feature profile, providing access to total and average times, as well as to all call/application profile, representing each single feature execution.

- **SCOOP_PROFILER_ACTION_PROFILE**: represents actions executed by a processor. This is the base class for feature call/application profiles and profiling profiles.

- **SCOOP_PROFILER_FEATURE_CALL_APPLICATION_PROFILE**: provides times and durations for a single feature call/application. It allows direct access to internal calls (call stack) and to all involved processors (caller processor, callee processor, locked processors).

- **SCOOP_PROFILER_PROFILING_PROFILE**: represents a profiling action. This might be contained in the list of calls made by a processor or in the call stack of a feature call/application.

- **SCOOP_PROFILER_HELPER**: helper class that provides facilities to calculate the duration and to convert them.

![Figure 4.4: BON diagram of profile abstraction](image-url)
4.6 Loader and Visitor Interaction

The interaction between loaders and visitors is not trivial, so we will demonstrate it with some examples, to understand the functioning of both. Visiting other events, not shown in this section, follows the same rules and proceeds in an analogue manner. All examples are taken from the default visitor and loader.

Loader Features

Let us first take a closer look at the operations provided by the loader. The Listing A.9 showing the main loop of the default loader can be found in the appendix.

Calling feature continue, Listing 4.1, makes the loader remove the current event from the event flow of the processor specified with \texttt{a_processor}. This is called to advance to the next event.

Feature delay enables the visitor to temporarily stop the reading of a certain processor’s events, as shown in Listing 4.2. This leaves the responsibility to provide events in temporal order with the loader, it releases the visitor to maintain a waiting queue or similar structure. Deactivating an event flow is necessary to maintain the logical succession of events and therefore avoid the swapping of events. This could happen if two events have the same time and thus could be given in any order by the loader.

Finally the resume feature, in Listing 4.3, enables the visitor to reactivate a previously stopped event flow. It is worth noting that events with a later timestamp can already have been processed, but this is fine, because the dependencies between processors are always guaranteed, i.e. the relative event flows are delayed, respectively resumed, maintaining a consistent state.
Implementation - Loader and Visitor Interaction

Listing 4.3: Eiffel: Loader: resume

```eiffel
resume (a_processor: INTEGER)

local
  l_item: SCOOP_PROFILER_FILES_DESERIALIZER

do
  l_item := hash.item(a_processor)
  if not (hpq.has(l_item) or l_item.after) then
    hpq.extend(l_item)
  end
end
```

Listing 4.4: Eiffel: Visitor: processor start

```eiffel
visit_processor_start (a_event: SCOOP_PROFILER_PROCESSOR_START_EVENT)
  -- Visit processor start.

do
  -- Set information
  find_processor (a_event.processor_id).set_start_time (a_event.time)

  -- Continue with events for this processor
  continue (a_event.processor_id)
end
```

Event Visitor Operation

Now that the three basic operations of a loader have been clarified, we show some examples that make use of them. We start with a very simple example and proceed explaining the dependency introduced by call and external call events.

Listing 4.4 shows the processing of the events representing the processor start. Line 5 finds the processor profile with a helper feature and sets the start time. Line 7 tells the loader that it can proceed to the next event, the current one has been processed correctly.

The same operation happens when visiting a call event, as shown in Listing 4.5. Lines 9 to 11 retrieve the objects referenced by the event. Lines 14 to 19 create a new feature call/application profile and set all the information. Line 22 adds it to the processor’s calls list. In the end, line 25 tells the loader to continue with the loading of events for this processor.

The processing of external calls events is a little more complex, it is shown in Listing 4.6. lines 8 to 11 prepare the required references. We can distinguish three cases:

- There is no external call in progress, so we can just register it (line 20) and update the processor queues, lines 21 to 23. We resume the processing of the called processor events and get rid of the current event in the caller (current) processor, afterwards we delay successive events from the current processor.
Implementation - Visualization

Listing 4.5: Eiffel: Visitor: visit feature call

```eiffel
define visit_feature_call (a_event: SCOOP_PROFILER_FEATURE_CALL_EVENT)
local
    c: SCOOP_PROFILER_CLASS_PROFILE
    f: SCOOP_PROFILER_FEATURE_PROFILE
    fc: SCOOP_PROFILER_FEATURE_CALL_APPLICATION_PROFILE
begin
    -- Find elements
    p := find_processor (a_event.processor_id)
    c := find_class (a_event)
    f := find_feature (a_event)
    -- Create feature call
    fc := profile.new_feature_call_application_profile
    fc.set_definition (f)
    fc.set_processor (p.id)
    fc.set_caller_processor (find_processor (a_event.caller_id))
    fc.set_call_time (a_event.time)
    fc.set_synchronous (a_event.synchronous)
    -- Add to calls
    calls.item (p.id).extend (fc)
    -- Continue with events for this processor
    continue (p.id)
end
```

- There is already an external call on the call stack and we have to wait for it to return, line 12. We delay the processing of the current and subsequent events for the current processor.
- There is already an external call registered and we have to wait for it to start (wait event on called processor), line 15. We delay the processing of the current and subsequent events for the current processor.

4.7 Visualization

In this section we describe the visual representation of a profile and discuss some technical details of the implementation.

Representation

EiffelVision is a library to build user interfaces and is not well suited to create diagrams. We decided to go for it because it allows the profiler to be used on every operating system and architecture supporting EVE. Figure 4.5 graphically represents the different profile abstractions and Table 4.2 wraps up all the classes involved in the visualization.

To display a profile we created a refinement of the abstraction described in Section 4.5, which enables us to create EiffelVision widgets. As this abstraction
Listing 4.6: Eiffel: Visitor: external call

```eiffel
visit_feature_external_call (a_event:
    SCOOP_PROFILER_FEATURE_EXTERNAL_CALL_EVENT)
-- Visit external call.
local
p: SCOOP_PROFILER_PROCESSOR_PROFILE
fc: SCOOP_PROFILER_FEATURE_CALL_APPLICATION_PROFILE
do
  -- Find elements
  p := find_processor (a_event . processor_id)
  if not stack.item (p.id).is_empty then
    fc := stack.item (p.id).item
  end
  if fc /= Void and then (fc.synchronous and fc.caller_processor.
    id = p.id and fc.processor.id /= p.id) then
    --| There is already a synchronous external call |--
    delay (p.id)
  elseif external_call.has (p.id) then
    --| We are already waiting for an external call |--
    delay (p.id)
  else
    --| Ok, register external call
    external_call.extend (a_event, p.id)
    resume (a_event . called_id)
    continue (p.id)
    delay (p.id)
  end
  end
```

inherits directly from the standard abstraction, we can build it with the available visitors. Listing 4.7 exemplifies the creation and loading of the EV profile abstraction.

For the integration in EVE we created a second refinement, built on top of the EV abstraction, to have access to the INTERFACE_NAMES class. This class dynamically translates the labels used as representation for profile elements, such as feature executions or profiling actions. Only the classes that need access to the localization facilities have been subclassed. The other components of the profile are the EV profile classes.

![Figure 4.5: Structure of profile abstractions](image)
Listing 4.7: *Eiffel*: Loading profile data into EV abstraction

```eiffel
profile : SCOOP_PROFILER_EV_APPLICATION_PROFILE

--- Application profile
build_profile

--- Create and populate 'profile'.
local
l_directory : DIRECTORY
l_loader : SCOOP_PROFILER_DEFAULT_LOADER
l_visitor : SCOOP_PROFILER_PARTIAL_EVENT_VISITOR

--- Prepare directory and profile
create l_directory.make ("<directory>*")
create profile.make
create l_loader.make_with_directory (l_directory)

--- Load the data using the "partial" visitor
l_loader.load (profile, create l_visitor.make_with_profile (profile))

--- Display processor rows information
main_container.extend (profile.widget)

--- Display summary table
main_container.extend (profile.grid_widget)
end
```

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Classes involved</th>
</tr>
</thead>
</table>
| Standard Profile     | SCOOP_PROFILER_APPLICATION_PROFILE  
|                      | SCOOP_PROFILER_PROCESSOR_PROFILE  
|                      | SCOOP_PROFILER_CLASS_PROFILE  
|                      | SCOOP_PROFILER_FEATURE_PROFILE  
|                      | SCOOP_PROFILER_ACTION_PROFILE  
|                      | SCOOP_PROFILER_FEATURE_CALL_APPLICATION_PROFILE  
|                      | SCOOP_PROFILER_PROFILING_PROFILE  
|                      | SCOOP_PROFILER_HELPER  
| EV Profile            | SCOOP_PROFILER_EV_APPLICATION_PROFILE  
|                      | SCOOP_PROFILER_EV_CLASS_PROFILE  
|                      | SCOOP_PROFILER_EV_FEATURE_PROFILE  
|                      | SCOOP_PROFILER_EV_FEATURE_CALL_APPLICATION_PROFILE  
|                      | SCOOP_PROFILER_EV_PROCESSOR_PROFILE  
|                      | SCOOP_PROFILER_EV_ACTION_PROFILE  
|                      | SCOOP_PROFILER_EV_PROFILING_PROFILE  
|                      | SCOOP_PROFILER_EV_HELPER  
|                      | SCOOP_PROFILER_EV_PROFILE  
|                      | SCOOP_PROFILER_EV_HOTSPOTS  
| Integration Profile  | EB_SCOOP_APPLICATION_PROFILE  
|                      | EB_SCOOP_FEATURE_CALL_APPLICATION_PROFILE  
|                      | EB_SCOOP_PROCESSOR_PROFILE  
|                      | EB_SCOOP_PROFILING_PROFILE  

Table 4.2: Classes involved in profile abstractions
Listing 4.8: Eiffel: Applying a zoom action

```eiffel
1 zoom_in
2   -- Zoom in.
3   do
4       lock_update
5       profile.set_zoom (profile.zoom * 2)
6       main_container.wipe_out
7       main_container.extend (profile.widget)
8       unlock_update
9   end
10 zoom_out
11    -- Zoom out.
12    do
13       lock_update
14       profile.set_zoom (profile.zoom / 2)
15       main_container.wipe_out
16       main_container.extend (profile.widget)
17       unlock_update
18   end
```

Caching and Redrawing

The first iteration of the EV drawing built the complete visualization diagram, including the whole internal call stack of each feature and wait-by-necessity spans. We tried to cache the whole diagram to support future redrawing (zoom, adding and removing elements, ...), but an update of an existing diagram took more time than redrawing it completely. This is due to EV redrawing all the objects which depend on the position of a modified object: an update of n widgets in a diagram is not linear, however a fresh redraw from left to right is linear with respect to the number of objects drawn.

It turned out that the best solution was to keep in memory only the key widgets (processor and feature labels) and reconstruct the rest of the structure every time. Internal calls are build on demand, cached, and discarded when drawing a fresh diagram, for instance by zooming.

Listing 4.8 shows how to implement zoom actions to redraw the EV profile, this is used in the graphical wizard in combination with a toolbar. The listing contains the two agents used.

Wizard

The existing profiler wizard has been extended with new steps to allow the user to select SCOOP related options. Following is a description of the involved classes:

- **EB_PROFILER_WIZARD_DATA_TYPE_STATE**: It lets the user select the type of data he wants to visualize: traditional profile data and SCOOP profile data are the two options. This step is injected in the existing profiler
wizard, right after the welcome screen. If the user chooses to continue with traditional data the existing wizard will be executed.

- **EB_PROFILER_WIZARD_DIRECTORY_STATE**: It lets the user choose the directory where the profile data is saved. The default directory where the profiler generates the data is proposed.

- **EB_PROFILER_WIZARD_DIRECTORY_ERROR_STATE**: This step is shown when there are problems reading the directory or the profile files.

- **EB_PROFILER_WIZARD_RUNS_STATE**: It lets the user choose the run he wants to visualize. The profiler saves the profile data for each run in a distinct subdirectory.

- **EB_PROFILER_WIZARD_OPTIONS_STATE**: It lets the user define visualization options. The interval to be profiled can be chosen with two range scrollers representing the start and end times of the interval. Moreover, the user can activate the hot spots analysis.

- **EB_SCOOP_PROFILE_WINDOW**: This class represents the final visualization window. It display the widgets and provides facilities to change the diagram (zoom).
Chapter 5

Evaluation

We will now evaluate our solution and point out some problems with the current implementation.

5.1 Demonstration Usage

In this section we demonstrate the use of the profiler with a couple of example programs, written in the context of the concurrency course at ETH in spring 2009 and inspired by synchronization problems described in [2].

All runs of both programs have been executed on the same system, namely in a virtual machine running Windows 7 and 1024 MB of RAM at disposal; the host operating system was Ubuntu Linux. The computer was equipped with an Intel Core2 Duo CPU at 2.66 GHz and 2560 MB of RAM.

Baboon Crossing

There is a deep canyon and a single rope that spans the canyon. Baboons can cross the canyon by swinging hand-over-hand on the rope, but if two baboons going in opposite directions meet in the middle, they will fight and drop to their deaths. Furthermore, the rope is only strong enough to hold 5 baboons. If there are more baboons on the rope at the same time, it will break. (The Little Book of Semaphores [2], pages 177-178)

With this first example we want to exemplify the use of the grid widget (Figure 5.1), to improve the profiled program. As we can see, there are already some hot spots highlighted for us, we try to remove the first one on the left.

Step 1 - Argument change

The cited hot spot shows the high number of calls for the direction feature of class BABOON. The calls originate from feature announce in class ROPE. As
direction is a callback on a separate object, it involves a SCOOP request and thus the collection of profiling events.

If we look at the code, we will find out that these calls can be removed completely, in the sense of separate calls, by passing a boolean argument to feature announce of class ROPE.

Listings 5.1 and 5.2 show the modifications for class BABOON. Listings 5.3 and 5.4 show the modifications for class ROPE.

We can see in Figure 5.2 that this small change improves the percentage of useful time of many features. The feature direction is not called anymore, causing less work for the SCOOP library (it was a separate call) and fewer events generated by the profiler.

Step 2 - Detachable mark

In the same manner we can improve the creation of objects, by passing references as detachable. As we do not want to call features during the creation, in this way the separate arguments do not get locked and the reference can be stored immediately, with no synchronization time.

Listings 5.5 and 5.6 show this simple modification to feature make of class BABOON. This modification will not be part of the Results section, because until recently the SCOOP compiler did not support detachable arguments.
Listing 5.1: **Eiffel**: Original BA-BOON

```eiffel
anounce (a_rope: separate ROPE)
   -- Announce.
   do
      io.put_string (out + "announcing\n")
      a_rope.announce (Current)
   end
```

Listing 5.2: **Eiffel**: Modified BA-BOON

```eiffel
announce (a_rope: separate ROPE)
   -- Announce.
   do
      io.put_string (out + "announcing\n")
      a_rope.announce (direction)
   end
```

Listing 5.3: **Eiffel**: Original ROPE

```eiffel
announce (a_baboon: separate BABOON)
   -- Each baboon should announce the rope first
   do
      if directions.is_empty then
         if baboons > 0 then
            changing := direction /= a_baboon.
            direction
         end
         direction := a_baboon.
         direction
      end
      directions.extend (a_baboon.
         direction)
   end
```

Listing 5.4: **Eiffel**: Modified ROPE

```eiffel
announce (a_direction: BOOLEAN)
   -- Each baboon should announce the rope first
   do
      if directions.is_empty then
         if baboons > 0 then
            changing := direction /= a_direction
         end
         direction := a_direction
      end
      directions.extend (a_direction)
   end
```
Listing 5.5: *Eiffel*: Original BA-BOON creation

```eiffel
make_with_rope (a_id: INTEGER; a_rope: separate ROPE)

- - Creation procedure.

  do
  id := a_id
  rope := a_rope
  create random.set_seed (id)
  random.forth
  direction := random_integer (1, 4) /= 1
  io.put_string (out + "brought to life\n")
  end
```

Listing 5.6: *Eiffel*: Modified BA-BOON creation

```eiffel
make_with_rope (a_id: INTEGER; a_rope: detachable separate ROPE)

- - Creation procedure.

  do
  id := a_id
  if attached (separate ROPE)
    rope := a_rope
  random.set_seed (id)
  random.forth
  direction := random_integer (1, 4) /= 1
  io.put_string (out + "brought to life\n")
  end
```

Step 3 - Factory

We try to make the creation process a little bit faster, by introducing factories for baboons. The root procedure creates a number of factories, that in turn create the baboons. Listings 5.7 and 5.8 reflect the changes to the root feature (feature `make` of class `BABOON_CROSSING`). You can find the new `BABOON_FACTORY` class in Listing A.10.

It is not a problem shown by the grid in Figure 5.2, but it helps demonstrating that the profiler might help to find performance problems in the architecture of the program. The profiler itself is not able to evaluate the structure of the program nor to propose a different architecture. We will find out in the next section whether factories are a winning approach or if they do not improve the overall performance.

Results

The changes motivated by our profiler have been useful, as reported in Table 5.1 and visualized in Graph 5.3.

Changing the arguments as described in step 1 let us improve the performance by around 2%. The use of factories to create the baboons, as proposed in step 2, increases the performance of another 4%. By doing two simple profiling steps we were able to get a 6% reduction on the overall run time.
Listing 5.7: Eiffel: Original Baboon Crossing

1 make
2    __ Creation procedure.
3 local
4     t_baboon: separate BABOON
5     i: INTEGER
6    do
7       create random.set_seed (max_baboons + 1)
8       from
9       i := 1
10      until
11     i > max_baboons
12    loop
13       create t_baboon.
14          make_with_rope (i, rope)
15     launch_baboon (t_baboon)
16     __ Code for random sleep
17     i := i + 1
18    end
19  end
20
21 launch_baboon(a_baboon: separate BABOON)
22     __ Launch the baboon.
23    do
24      a_baboon.live
25    end
26

Listing 5.8: Eiffel: Modified Baboon Crossing

1 make
2    __ Creation procedure.
3 local
4     t_factory: separate BABOON_FACTORY
5     i: INTEGER
6    do
7       create random.set_seed (max_baboons + 1)
8       from
9       i := 1
10      until
11     i > max_factories
12    loop
13       create t_factory.
14          make_with_rope (i, rope, max_baboons // max_factories)
15          launch_factory (t_factory)
16      i := i + 1
17    end
18  end
19
20 launch_factory(a_factory: separate BABOON_FACTORY)
21     __ Launch the factory.
22    do
23      a_factory.live
24    end

<table>
<thead>
<tr>
<th>Run</th>
<th>Baboon Crossing</th>
<th>normal</th>
<th>arguments</th>
<th>arguments, factory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>15641 ms</td>
<td>15263 ms</td>
<td>14311 ms</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>15500 ms</td>
<td>15280 ms</td>
<td>14530 ms</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>15485 ms</td>
<td>15171 ms</td>
<td>15125 ms</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>15625 ms</td>
<td>15344 ms</td>
<td>14483 ms</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>15562 ms</td>
<td>15218 ms</td>
<td>14922 ms</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>77813 ms</td>
<td>76216 ms</td>
<td>73371 ms</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td>15563 ms</td>
<td>15243 ms</td>
<td>14764 ms</td>
</tr>
<tr>
<td>Stddes</td>
<td></td>
<td>71 ms</td>
<td>69 ms</td>
<td>337 ms</td>
</tr>
<tr>
<td>Effect</td>
<td></td>
<td>-2%</td>
<td>-6%</td>
<td>-6%</td>
</tr>
</tbody>
</table>

Table 5.1: Baboon Crossing execution times
Barber Shop

A barber shop consists of a waiting room with n chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. (*The Little Book of Semaphores* [2], page 127)

In this second example we look at the diagram widget, to see how customers can be better served in a barber shop.
busy all the time and the customer processors are idle, waiting for synchronization, until the barber is free. We want to alleviate this bottleneck by introducing a second barber in the shop. We expect an important improvement.

![Figure 5.5: Barber Shop profile 2](image)

We compare the versions with one and two barbers, running each of them 5 times and recording the run times. The results are listed in Table 5.2 and can be seen visually in Graph 5.6.

<table>
<thead>
<tr>
<th>Run</th>
<th>Barber Shop</th>
<th>normal</th>
<th>two barbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32625 ms</td>
<td>16344 ms</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>31844 ms</td>
<td>16296 ms</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>31688 ms</td>
<td>16328 ms</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>32250 ms</td>
<td>16500 ms</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>31828 ms</td>
<td>16360 ms</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>160235 ms</td>
<td>81828 ms</td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>32047 ms</td>
<td>16366 ms</td>
<td></td>
</tr>
<tr>
<td>Stddev</td>
<td>154 ms</td>
<td>32 ms</td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>-</td>
<td>-49%</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Barber Shop execution times

As expected, the performance of the program increased by nearly 50%. The diagram shows it clearly and understandably: we can easily identify and pair the features running in the customer processors and those executed by the barbers.
5.2 Observer Effect

In this section we would like to investigate more about the effects on the program caused by the use of the profiler. In other words we try to understand how much the results are influenced by collecting and serializing the events.

Writing events to disk is certainly the main issue because the processor has to stop the normal work flow, do input/output operations, and flush all the buffered events. As these profiling actions are orders of magnitude more expensive than collecting events, we only record these operations in events, so that they can be shown transparently to the user. The user can overcome the problem by setting a very large collection buffer, profiling actions will thus be delayed and executed later.

Memory has an upper bound given by the buffer size, plus a system-wide constant for storing the profiling information, which is in turn bounded by the number of separate classes and features.

The second, more subtle, issue is the influence of event generation and IDs to names conversion. We devised a simple test, which consists in measuring the total run time of two example programs, taken without modifications from the SCOOP examples included in EVE.

We first run the programs, recording only the creation and removal times of the processors (columns "normal" in Table 5.3). In a second run (columns "collection") we enable the whole event generation and measure again the creation and removal times for processors. For one example we did a third run (column "collection, serialization") measuring also the time taken to write events to disk. To have a meaningful result, each test is run 10 times.

From these little tests we devise that collecting events influences the run time by 2-4%, serializing the events in addition to collecting them takes 70-72%
additional time. Effect is calculated on normal run time. Table 5.3 reports all the measured times and Graph 5.7 depicts the results graphically.

<table>
<thead>
<tr>
<th>Run</th>
<th>Barber Shop</th>
<th>Baboon Crossing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>normal</td>
<td>collection</td>
</tr>
<tr>
<td>1</td>
<td>31969 ms</td>
<td>36297 ms</td>
</tr>
<tr>
<td>2</td>
<td>32141 ms</td>
<td>32717 ms</td>
</tr>
<tr>
<td>3</td>
<td>31968 ms</td>
<td>32469 ms</td>
</tr>
<tr>
<td>4</td>
<td>31889 ms</td>
<td>33141 ms</td>
</tr>
<tr>
<td>5</td>
<td>31922 ms</td>
<td>33483 ms</td>
</tr>
<tr>
<td>6</td>
<td>31953 ms</td>
<td>32765 ms</td>
</tr>
<tr>
<td>7</td>
<td>32063 ms</td>
<td>33250 ms</td>
</tr>
<tr>
<td>8</td>
<td>31829 ms</td>
<td>32703 ms</td>
</tr>
<tr>
<td>9</td>
<td>32016 ms</td>
<td>33297 ms</td>
</tr>
<tr>
<td>10</td>
<td>32186 ms</td>
<td>33359 ms</td>
</tr>
</tbody>
</table>

Mean: 31994 ms 33348 ms 15068 ms 15362 ms 26256 ms
Stddev: 111 ms 1090 ms 231 ms 80 ms 1300 ms
Effect: +4% - +2% +74%

Table 5.3: Analysis of probe effect

5.3 Problems

We briefly discuss some problems that came up during the project. These are mainly implementation problems that do not depend on our work, and that we could not fix on our own.

Timestamps Resolution

The best resolution that we could get using a DATE_TIME object is 1 millisecond. We first tried to tackle this problem by using a C external that could lead to better resolution (at least microseconds), but the suggested function was not working the way we expected and we did not have documentation at hand to find out what was wrong.

SCOOP Compiler

The Eiffel standard contemplates detachable separate arguments and the type checker allows them, but the SCOOP compiler is not currently able to generate specific client and proxy code for features with detachable arguments. Instead, it generates normal code as if the detachable keyword were absent.

This is not a problem from the correctness point of view, but it accounts for performance loss from the SCOOP side, as it obliges the scheduler to lock arguments that could otherwise be considered as simple references. Creation
procedures could, in many examples, be run without having to wait and the overall effect might be quite noticeable.

The problem is already known and there is already an open issue for this. Nevertheless, it demonstrates that the profiler can be of help to measure the strength of the underlying implementation and improve it.

**EiffelVision**

EiffelVision, being a graphical user interface library, is not well suited for drawing diagrams and its limits are evident when drawing the large amount of labels and boxes required by the EV profile abstraction.

We demand attention on the fact that most interaction tests were conducted without finalizing the project and run on a virtual machine environment which could not harness all the available CPU and graphics card power. Besides this consideration, some effort is necessary to have a simple but powerful graphics library that can be used on multiple platforms.
Chapter 6

Developers Guide

In this chapter we provide a guide for developers who want to know more about the functioning of the SCOOP profiler or want to work on the implementation.

6.1 Configuration

Checkout the source from the SCOOP Subversion repository [18]. The code related to the profiler is in the folder $ISE_LIBRARY/scoopli/profiler.

Listing 6.1 shows the configuration options for the profiler with default values, contained in class SCOOP_LIBRARY_CONSTANTS. Here is the description of each option:

- **Enable_profiler**: whether to enable the profiler at all.
- **Enable_log**: whether to enable logging.
- **Profile_collector_buffer**: number of events to queue before writing to disk.
- **Profile_file_extension**: extension to apply to all profile files. Make sure this is not used by other files, as the wizard will check this extension for loading the profile data.
- **Profile_log_extension**: extension for the log files.
- **Information_file_name**: name of the file where to put class and feature names. This file will reside in the EIFGENs folder.
- **Information_file_extension**: extension of the information file.
- **Profile_directory**: name of the directory where to put the collected profile data. This directory resides in the target folder, inside the EIFGENs folder.
Listing 6.1: Eiffel: Profiler configuration options

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable_profiler</td>
<td>BOOLEAN</td>
<td>Whether to enable the profiler</td>
</tr>
<tr>
<td>Enable_log</td>
<td>BOOLEAN</td>
<td>Whether to enable log files</td>
</tr>
<tr>
<td>Profile_collector_buffer</td>
<td>INTEGER</td>
<td>Number of events to buffer before writing to disk (per processor)</td>
</tr>
<tr>
<td>Profile_file_extension</td>
<td>STRING</td>
<td>Extension for profile file</td>
</tr>
<tr>
<td>Profile_log_extension</td>
<td>STRING</td>
<td>Extension for log file</td>
</tr>
<tr>
<td>Information_file_name</td>
<td>STRING</td>
<td>Information file name</td>
</tr>
<tr>
<td>Information_file_extension</td>
<td>STRING</td>
<td>Extension for information file</td>
</tr>
<tr>
<td>Profile_directory</td>
<td>STRING</td>
<td>Profile directory</td>
</tr>
</tbody>
</table>

6.2 Profile Information

The profile information is built during the compilation of the program, in feature `build_profile_information` of class `DEGREE_SCOOP`. Related classes can be found in the cluster `profiler/information` in the `scoopli` library.

At run time each processor keeps a reference to the profile information, which is deserialized in feature `read_profile_information` of class `SCOOP_SCHEDULER`. The run time IDs and names are accessed through the features `feature_*_of`, `class_*_of`, `processor_id_of` in class `SCOOP_PROFILER_COLLECTOR`.

6.3 Events

The event classes can be found in cluster `profiler/events` in the `scoopli` library. They are created by calling the respective `collect_*` features from class `SCOOP_PROFILER_COLLECTOR`.

The same class is responsible for serializing periodically the events, involved features are: `setup` (called once at creation), `prepare_serialization`, `serialize`, `cleanup_serialization`, and `file_name`.

The visitors and loader related classes can be found in clusters `profiler/visitors` and `profiler/loaders`.

6.4 Profile Abstractions

Two abstractions have been developed to represent a profile. The base class for the standard abstraction is class `SCOOP_PROFILER_APPLICATION_PROFILE` which is also used as factory to create class, feature, processor, and profiling profiles.
The entry point for developing the EV profile abstraction, and factory to instantiate all EV profiles, is class `SCOOP_PROFILER_EV_APPLICATION_PROFILE`. These classes provide a widget feature to be used for visualization with EiffelVision.

For the abstraction to integrate the visualization into EVE the class is `EB_SCOOP_APPLICATION_PROFILE`. These classes contain a reference to the class `INTERFACE_NAMES` in `interface_names` library, which is used as a central place to define all the strings used in the graphical user interface.

### 6.5 Profiler Wizard

The classes related to the profiler wizard can be found in cluster `interface/new_graphical/profile_tool` defined in target `bench` of the EVE project.

The EV profile abstraction classes are located in subcluster `ev`, while the integration abstraction can be found in subcluster `integration`, where you can also find the class `EB_SCOOP_PROFILE_WINDOW` representing the visualization window.

All the new steps have been added to the subcluster `wizard`. The entry point for changes to the wizard is class `EB_PROFILER_WIZARD_INITIAL_STATE`, where the step to select the profile data type is inserted. The two branches of the profiler wizard start in class `EB_PROFILER_WIZARD_DATA_TYPE_STATE`. 
This User Guide explains how to enable the SCOOP profiler and how to use the profiler wizard provided by EVE.

The profiler is already included in the last release of EVE with SCOOP, which you can find at [18] in the download section. We assume EVE to be correctly installed. You can follow the EVE setup guide that can be found in the documentation section.

7.1 Enabling the Profiler

The SCOOP profiler is enabled in the ECF. To enable the profiling of a program, it is sufficient to add the following line to the ECF file of the project: `<setting name="scoop_profile" value="true"/>`.

The profiler buffer size can be configured in the ECF with this line: `<setting name="scoop_profile_buffer" value="0">`, this means that you can fine tune how many events per processors are buffered before writing them to disk. If you do not specify the setting, the default hardcoded value will be used. For a short running program use a high number so that the events are written to disk only once; for a program that does not terminate (see some examples in the scoop folder) use 0 or a very low number.

Table 7.1 summarizes the available ECF settings.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>scoop_profile</td>
<td>- true: enables the profiler</td>
</tr>
<tr>
<td></td>
<td>- false: disables the profiler</td>
</tr>
<tr>
<td>scoop_profile_buffer</td>
<td>- integer: defines the size of the collector</td>
</tr>
<tr>
<td></td>
<td>buffer</td>
</tr>
</tbody>
</table>

Table 7.1: Overview of ECF settings.
7.2 Running the wizard

After compiling and running the program, start the profiler wizard by accessing the Tools menu and selecting Profiler. Choose Next at the welcome step, as shown in figure 7.1.

![Menu selection and welcome screen](image)

Figure 7.1: Starting the profiler

The second step will present you with the possibility to choose between the traditional profile data and the SCOOP profile data. Choose the second option and click Next, as shown in figure 7.2.

![Data type selection screen](image)

Figure 7.2: Profiler wizard: data type selection

The next step will propose you the standard directory where the profile data resides. You usually have the right directory already in the field and should click Next, as shown in figure 7.3.

If you ran the program more than once, the next step will ask you which run you want to show. Choose the right starting time and click Next, as shown in figure 7.4.

The next step lets you choose some options for displaying the profile data:
Figure 7.3: Profiler wizard: directory selection

Figure 7.4: Profiler wizard: run selection
• Start and stop time: this will restrict the time span to be displayed. Only features that start in the given time span are shown. If a feature does not end in the given range it will be marked as incomplete and, for visualizing, the possibly missing times (application and/or return time) will be set to the end of the chosen time span.

• Enable hotspots: this option will cause the wizard to show simple hot spots in the grid view.

Choose the right options and click Finish, as show in figure 7.5.

![Profiler wizard: options](image)

Figure 7.5: Profiler wizard: options

### 7.3 Visualizing Results

The feature calls, per processor, are displayed in the upper part of the window, showing waiting intervals (yellow), execution intervals (green), and profiling intervals (red). Clicking on a feature displays a subrow showing the internal calls and wait condition trials.

Dependencies between calls are also displayed. The processors that are being locked by a feature (in the synchronization interval) are highlighted when the mouse is moved on the respective interval. When the mouse is moved on a call event (red vertical bar), the respective external calls are highlighted. Moreover the internal calls are shown on the caller processor’s diagram.

In the lower part of the window a grid is displayed with all the numeric information about the profiled time span:

• Name: name of the class and feature.
• Number of calls: total number of calls.
• Wait condition tries: total and average number of wait condition tries.
• Queue time: total, average, and standard deviation time until the processor is ready to execute the call.

• Synchronization time: total, average, and standard deviation time spent waiting for locks and trying the wait conditions.

• Execution time: total, average, and standard deviation time executing the feature’s body.

• Percentage of useful time: percentage of average execution time used not waiting for external calls or profiling operations.

Hot spots are highlighted in the grid, using simple heuristics. The maximum number of calls, the maximum average number of wait condition trials, and the minimum percentage of useful time are strongly highlighted: here is where the programmer should strive for an improvement.

Other spots are lightly pointed out if they do not represent the best possible result. An average of 0 or 1 wait condition trials is considered optimal and so is a percentage of useful time of 100% and queue, synchronization, and execution times of 0ms.

Figure 7.6 shows the profiler window for an example program.
Chapter 8

Future Work

In this chapter we present some ideas about future work possibilities. The proposed work packages have been considered as minor priority in this work or were found interesting additions during the development and use of the profiler.

8.1 Static Metrics

To complete the facilities integrated into EVE for profiling SCOOP programs, it would be useful to add SCOOP metrics to the metrics tool. Some interesting ones would be:

- Number of features taking separate arguments
- Number of separate arguments per separate feature
- Number of separate pre-/postcondition clauses
- Number of lock passing occurrences
- Number of classes that are used as base classes for separate types
- Number of separate attributes

8.2 Dynamic Metrics

We found some other interesting dynamic metrics, that have not been added yet. Here is some example:

- Number of processors in the system over time: for this it is necessary to check how processors get disposed. At the time of writing processors do not get disposed as soon as they are not needed.
• Number of objects per processor over time: it would be interesting to know how many objects live on each processor and how busy each processor is depending on the number of objects it has to handle and on the type of these objects.

• Number of actually separate postconditions, only known at runtime.

• Number and nature of calls per separate feature (separate calls, non-separate calls, separate callbacks).

8.3 Events

The current implementation of the `SCOOP_PROFILER_COLLECTOR` flushes the events based on a constant defined in `SCOOP_LIBRARY_CONSTANTS`, which can be overridden in the ECF file (see Chapter 7). A processor that does not produce a lot of events is flushed just a few times.

The processor handling the root object will just be flushed once, and it might happen at the end of the program execution. If the program has still a waiting thread (in some examples there are actors that do not stop), but it has already produced the expected results, it is normal to kill the program to stop debugging.

Because of this, the external call event for the root feature might not have been written to disk, whereas its call counterpart might have been written without problems, by a more active processor. This behavior might cause problems when viewing the results, because when the visitor encounters the call event for the root feature, it waits until the external call event appears; this will never happen and the partial profile is not shown, even if the processing went well.

We can think of a new way of handling the writing of events to disk, by means of a separate thread and the observer pattern, so that each processor generates the events and a configurable observer can write them to disk, without interfering too much with the normal flow of the program. This is possible because after a processor has generated the events it does not need to wait for file operations. Some interference would still exist, because the observer processor has to reserve some resources and do input/output operations.

8.4 Data Serialization

A lot of information is saved when an event is flushed to disk. One could refine the events abstraction to restrict the amount of data needed by each event. Moreover, one could investigate other serialization libraries and choose one which produces smaller files.

Dadle [13] is an alternative object serialization library for Eiffel. We did not have a deeper look at it, mainly because we did not want to add a dependency to our profiler, but it might be a promising library and it is already used in EVE in Ebbro (Eiffel Object Browser, see [14]).
8.5 Graphical Wizard

The EV profile abstraction has to be reviewed and optimized in terms of speed. Despite building all the boxes on demand, the structure of boxes and labels could be improved. That said, EiffelVision might not be the best library for drawing diagrams, so when a small cross-platform graphic library is available it might be interesting to port the abstraction.

The profiler wizard is not able to extend the range of the loaded data, even if the abstractions and the diagram allow us to do it. One possible extension is to modify the implementation of the loaders and visitors, so that a user can interactively decide to discard some data or to extend the profiled interval.

It would be nice to give a meaningful name to each run, for example "First working version, option xyz". It is much more usable than selecting a profile based on the date when the program was profiled.

8.6 Profile Analysis

In the current implementation a simple hotspot highlighting system is used. This mechanism is based on heuristics like the maximum number of calls and the smallest percentage of useful time. It would be interesting to implement a more complex analysis system that can highlight more complex hot spots and give suggestions on how to improve the performance of the program.
Chapter 9

Conclusions

We end this thesis with some considerations about the project, the goals that we achieved, and some personal comments.

9.1 Critique

Despite the clear goals, the path to get to a robust solution was not obvious. We set precise goals and worked hard striving for a very qualitative outcome. We think to have accomplished all the goals and briefly discuss them in the following list.

- *Design a profiler framework, and visualization tools*: As documented in Chapter 3, all the concepts are distinct and this reflects to the implementation, described in Chapter 4.

- *Clearly document the design choices and metrics*: Building on the related work described in Chapter 2, Chapter 3 explains the different metrics used.

- *Implement and integrate the profile generation and wizard into EVE*: All the code is integrated into EVE and is part of the last releases that include SCOOP.

- *Properly document and test all the code*: All the code has been tested with a few examples from the ones included in EVE. The attribute, feature, class names are as self-explanatory as possible and we tried to put comments to clarify what particular pieces of code do.

- *Profile a few example applications and their improved versions in term of performance*: The results of running the profiler on different programs and the originated improvements can be found in Chapter 5.

- *File eventual bugs in the SCOOP compiler and library*: Some small problems with the SCOOP compiler and the examples included in EVE have been filed and are in the issue list of the respective maintainers.
9.2 Personal Remarks

During this thesis I had the possibility to work on the whole project by myself, conducting research to support decisions and evaluating the implementation. Moreover, I had the possibility to look into the SCOOP compilation process and the inner workings of the Eiffel compilation.

I am confident that SCOOP, along with Eiffel, will go a long way and help lowering the learning curve of concurrent programming. I hope that my work will help in this long term effort and be used to build more and more great SCOOP applications.
Appendix A

Listings

A.1 Gathering of profile information

```eiffel
1  -- In class DEGREE_SCOOP, feature build_profile_information.
2  if attached (CLASS_TYPE) system.class_types.item (i) as l_type AND
3     then l_type.associated_class.group.name.is_equal (
4       SCOOP_SYSTEM_CONSTANTS).Scoop_override_cluster_name) THEN
5     create l_name.make_from_string (l_type.associated_class.
6       name_in_upper)
7     -- Remove scoop prefix
8     if l_type.associated_class.name_in_upper.starts_with (
9       SCOOP_SYSTEM_CONSTANTS).Scoop_proxy_class_prefix) THEN
10    l_name.remove_head (SCOOP_SYSTEM_CONSTANTS).
11       Scoop_proxy_class_prefix.count)
12  end
13  -- Do not profile scoop starter class
14  if not l_name.is_equal (SCOOP_SYSTEM_CONSTANTS).
15     Scoop_starter_class_name) THEN
16     create l_class_info.make
17     l_class_info.set_name (l_name)
18     information.classes.extend (l_class_info, l_type.type_id)
19     -- Produce profile information for features
20     See listing A.2
21  end
22
Listing A.2: Eiffel: Feature information gathering
```

```eiffel
1  -- In class DEGREE_SCOOP, feature build_profile_information.
2  l_feature := l_type.associated_class.feature_table.
3  -- Process feature only if it is written in SCOOP classes
4  if l_feature.written_class.group.name.is_equal (
5     SCOOP_SYSTEM_CONSTANTS).Scoop_override_cluster_name) THEN
6  -- Create feature information
7     create l_feature_info.make
8     -- Get feature name
9     create l_name.make_from_string (l_feature.e.feature.name)
```

58
A.2 Collection of events

Listing A.3: Eiffel: Collection of events by scheduler

```eiffel
-- In class SCOOP_SCHEDULER
execute_routine (a_caller: SCOOP_SEPARATE_CLIENT;
    a_requested_processors: TUPLE [SCOOP_PROCESSOR]; a_routine:
    ROUTINE [SCOOP_SEPARATE_TYPE, TUPLE]; a_wait_condition:
    FUNCTION [SCOOP_SEPARATE_CLIENT, TUPLE, BOOLEAN];
    a_separate_postcondition, a_non_separate_postcondition:
    ROUTINE [SCOOP_SEPARATE_CLIENT, TUPLE])
    -- Create routine request for 'a_routine' and add it to list of
    -- routine requests.
    -- When request is ready for execution, i.e. all requested
    -- processors are locked and 'a_wait_condition' holds, execute
    -- 'a_routine'.
    do
        -- [...] Setup code omitted
        -- SCOOP PROFILE
        if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and
            profile_information.is_profiling_enabled then
            scoop_profiler_target ?= a_routine.target
        end
        -- [...] Code for processor requests omitted
        -- SCOOP PROFILE
        if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and
            scoop_profiler_target /= Void then
            scoop_profiler_target.processor_.profile_collector.
            collect_feature_wait (a_routine, effectively_requested_processors)
```
Listing A.4: Eiffel: Collection of events by proxy

```eiffel
-- In class SCOOP_SEPARATE_PROXY
scoop_[a]_synchronous_execute (a_caller_: SCOOP_SEPARATE_TYPE;
  a_routine: ROUTINE [ANY, TUPLE])
  -- Ask handling processor to execute 'a_routine'.
  require
    a_caller_non_void: a_caller_ /= void
    a_caller_processor_non_void: a_caller_.processor_ /= void
    a_routine_non_void: a_routine /= void
  local
    scoop_collector: SCOOP_PROFILER_COLLECTOR
  do
    if processor_ = void then set_processor_ (a_caller_.processor_)
      end
    if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and processor_.profile_collector /= Void then
```

```eiffel
  if are_requested_processors_locked then
    -- SCOOP PROFILE
    if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and
      scoop_profiler_target /= Void then
      scoop_profiler_target.processor_.profile_collector.
        collect_feature_application (a_routine)
      a_routine.call([])
      scoop_profiler_target.processor_.profile_collector.
        collect_feature_return (a_routine)
    else
      a_routine.call([])
    end
    -- [...] Code for postconditions omitted
  else
    create request.make (a_caller_, requested_processors, 
      a_routine, a_wait_condition)
    routine_requests_mutex.lock
    routine_requests.extend (request)
    if routine_requests.after then
      routine_requests.start -- necessary to make sure that
        scheduler thread take new request into account
      end
    routine_request_update.set
    routine_requests_mutex.unlock
    request.ready_for_execution.wait_one -- wait until routine
      request is ready for execution (locks acquired and wait
      condition holds)
    a_caller_.processor_.locked_processors_push ( 
      requested_processors)
    -- SCOOP PROFILE
    if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and
      scoop_profiler_target /= Void then
      scoop_profiler_target.processor_.profile_collector.
        collect_feature_application (a_routine)
      a_routine.call([])
      scoop_profiler_target.processor_.profile_collector.
        collect_feature_return (a_routine)
    else
      a_routine.call([])
    end
    -- [...] Code for postconditions, and locks release omitted
  end
```

---

```
Listing A.4: Eiffel: Collection of events by proxy

15 if are_requested_processors_locked then
16   -- SCOOP PROFILE
17   if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and
18      scoop_profiler_target /= Void then
19      scoop_profiler_target.processor_.profile_collector.
        collect_feature_application (a_routine)
20      a_routine.call([])
21      scoop_profiler_target.processor_.profile_collector.
        collect_feature_return (a_routine)
22   else
23      a_routine.call([])
24   end
25   -- [...] Code for postconditions omitted
26 else
27    create request.make (a_caller_, requested_processors, 
      a_routine, a_wait_condition)
28    routine_requests_mutex.lock
29    routine_requests.extend (request)
30    if routine_requests.after then
31      routine_requests.start -- necessary to make sure that
          scheduler thread take new request into account
32      end
33    routine_request_update.set
34    routine_requests_mutex.unlock
35    request.ready_for_execution.wait_one -- wait until routine
      request is ready for execution (locks acquired and wait
      condition holds)
36    a_caller_.processor_.locked_processors_push ( 
      requested_processors)
37    -- SCOOP PROFILE
38    if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and
59 if are_requested_processors_locked then
60   -- SCOOP PROFILE
61   if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and
62      scoop_profiler_target /= Void then
63      scoop_profiler_target.processor_.profile_collector.
        collect_feature_application (a_routine)
64      a_routine.call([])
65      scoop_profiler_target.processor_.profile_collector.
        collect_feature_return (a_routine)
66   else
67      a_routine.call([])
68   end
69   -- [...] Code for postconditions omitted
70 else
71    create request.make (a_caller_, requested_processors, 
52    scoop_collector : SCOOP_PROFILER_COLLECTOR
53      processor_ = void
      set_processor_ (a_caller_.processor_)
74    end
75    if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and processor_.
```
scoop_collector := processor_.profile_collector

-- SCOOP PROFILE: Collect call (always)
scoop_collector.collect_feature_call (a_routine, a_caller_.
    processor_, a_caller_.processor_.
    synchronous_processors_has (processor_))

end

if a_caller_.processor_.synchronous_processors_has (processor_)
    then
    if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and
    scoop_collector != Void and then not scoop_collector.
        has_separate_arguments (a_routine) then
        -- SCOOP PROFILE: This doesn’t go through the scheduler,
        collect wait/application{return here
    scoop_collector.collect_feature_wait (a_routine, create (LINKED_LIST [SCOOP_PROCESSOR]).make)
    scoop_collector.collect_feature_application (a_routine)
    a_routine.call ([]) -- Execute ‘a_routine’ synchronously.
    scoop_collector.collect_feature_return (a_routine)
   end
   else
   a_routine.call ([]) -- Execute ‘a_routine’ synchronously.
   end
   else
   processor_.[a]synchronous_execute (a_caller_, a_routine)
end

Listing A.5: Eiffel: Collection of wait condition events

-- In class SCOOP_ROUTINE_REQUEST
wait_condition_satisfied : BOOLEAN

-- Is wait condition satisfied?
do
    if wait_condition /= void then
        -- SCOOP PROFILE
        if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and attached {
            SCOOP_SEPARATE_TYPE} routine target as
            scoop_profile_target and then scoop_profile_target.
            processor_.profile_collector /= Void then
                scoop_profile_target.processor_.profile_collector.
                    collect_wait_condition (routine)
            end
        wait_condition.call ([])
        Result := wait_condition.last_result
    else
        Result := True
    end
end

Listing A.6: Eiffel: Collection of processor events

-- In class SCOOP_SCHEDULER
new_processor : SCOOP_PROCESSOR

-- Create new processor and return reference to it.
do
    new_processor
    -- SCOOP PROFILER
    if profile_information = Void then
        read_profile_information
    end
    create Result.make (Current)
    processors_mutex.lock
    processors.extend (Result)
}
--- SCOOP_PROFILER
if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and
    profile_information.is_profiling_enabled then
    Result.profile_collector.collect_processor_add
end
processors_mutex.unlock

remove_processor (a_processor: SCOOP_PROCESSOR)
--- Remove 'a_processor' from list of processors.
require
    a_processor_not_void: a_processor /= void
do
    processors_mutex.lock
    processors.start
--- SCOOP_PROFILER
if (SCOOP_LIBRARY_CONSTANTS).Enable_profiler and
    profile_information.is_profiling_enabled then
    a_processor.profile_collector.collect_processor_end
end
processors.prune (a_processor)
if processors.is_empty then
    all_processors_finished.set
end
processors_mutex.unlock

A.3 Loaders and Visitors

Listing A.7: Eiffel: Class: Loader

defered class
    SCOOP_PROFILER_LOADER

feature -- Access
    start, stop, min, max: DATE_TIME

feature -- Status setting
    set_min (a_min: like min)
    -- Set 'min' to 'a_min'.
    require
        min_greater_equal_start: a_min /= Void and then a_min >=
            start
    do
        min := a_min
    ensure
        min_set: min = a_min
    end

    set_max (a_max: like max)
    -- Set 'max' to 'a_max'.
    require
        max_less_equal_stop: a_max /= Void and then a_max <= stop
    do
        max := a_max
    ensure
        max_set: max = a_max
set_progress_action (a_action: like progress_action)
  
  do
    progress_action := a_action
  ensure
    progress_action_set: progress_action = a_action
  end

load (a_profile: SCOOP_PROFILER_APPLICATION_PROFILE; a_visitor: SCOOP_PROFILER_EVENT_VISITOR)
  
  require
    profile_not_void: a_profile /= Void
    visitor_not_void: a_visitor /= Void
  deferred

feature {SCOOP_PROFILER_EVENT_VISITOR} -- Basic operation
  
  delay (a_id: INTEGER)
    
    require
      id_positive: a_id > 0
    deferred

resume (a_id: INTEGER)
  
    require
      id_positive: a_id > 0
    deferred

continue (a_id: INTEGER)
  
    require
      id_positive: a_id > 0
    deferred

feature (NONE) -- Implementation
  
  progress_action: PROCEDURE [ANY, TUPLE [DATE_TIME]]
    
    Reference to progress action

invariant
  times_ok: (start /= Void and stop /= Void) implies start <= stop
  limits_ok: (min /= Void and max /= Void) implies min <= max
end

Listing A.8: Eiffel: Class: Event visitor
loader: SCOOP_PROFILER_LOADER
-- Actual loader, this reads the events from a source

feature — Status setting
set_loader (a_loader: like loader)
-- Set 'loader' to 'a_loader'.
require
  loader_not_void: a_loader /= Void
do
  loader := a_loader
ensure
  loader_set: loader = a_loader
end

feature (SCOOP_PROFILER_LOADER) — Basic operation
cleanup
do
end

feature (SCOOP_PROFILER_EVENT) — Visitor
visit_processor_start (a_event: SCOOP_PROFILER_PROCESSOR_START_EVENT)
-- Visit processor start event.
require
  event_not_void: a_event /= Void
do
end
visit_processor_end (a_event: SCOOP_PROFILER_PROCESSOR_END_EVENT)
-- Visit processor end event.
require
  event_not_void: a_event /= Void
do
end
visit_profiling_start (a_event: SCOOP_PROFILER_PROFILING_START_EVENT)
-- Visit profiling start event.
require
  event_not_void: a_event /= Void
do
end
visit_profiling_end (a_event: SCOOP_PROFILER_PROFILING_END_EVENT)
-- Visit profiling end event.
require
  event_not_void: a_event /= Void
do
end
visit_feature_external_call (a_event: SCOOP_PROFILER_FEATURE_EXTERNAL_CALL_EVENT)
-- Visit external call event.
-- (separate call, generated on caller processor)
require
  event_not_void: a_event /= Void
do
end
visit_feature_call (a_event: SCOOP_PROFILER_FEATURE_CALL_EVENT)
    -- Visit feature call event.
    -- (separate call, generated on called processor)
    require
    event_not_void: a_event /= Void
    do
    end

visit_feature_wait (a_event: SCOOP_PROFILER_FEATURE_WAIT_EVENT)
    -- Visit feature sync event.
    require
    event_not_void: a_event /= Void
    do
    end

visit_feature_application (a_event: SCOOP_PROFILER_FEATURE_APPLICATION_EVENT)
    -- Visit feature application event.
    require
    event_not_void: a_event /= Void
    do
    end

visit_feature_return (a_event: SCOOP_PROFILER_FEATURE_RETURN_EVENT)
    -- Visit feature return event.
    require
    event_not_void: a_event /= Void
    do
    end

visit_feature_wait_condition (a_event: SCOOP_PROFILER_FEATURE_WAIT_CONDITION_EVENT)
    -- Visit wait condition try event.
    require
    event_not_void: a_event /= Void
    do
    end

Listing A.9: Eiffel: Loader: load

load (a_profile: like profile; a_visitor: SCOOP_PROFILER_EVENT_VISITOR)
    -- Load events into 'a_profile' using 'a_visitor'.
local
    l_loader: SCOOP_PROFILER_FILES_DESERIALIZER
    l_item: SCOOP_PROFILER_EVENT
    l_current: DATE_TIME
    l_list: ARRAYED_LIST [SCOOP_PROFILER_FILES_DESERIALIZER]
do
    profile := a_profile
    a_visitor.set_loader (Current)
    -- Start loaders
    l_list := hash.linear_representation
    l_list.do_all (agent (SCOOP_PROFILER_FILES_DESERIALIZER).start)
    -- Create priority queue
    create hpq.make (l_list.count)
from
  hpq.append (l_list)
until
  hpq.is_empty
loop
  -- Remove loader from queue
  l_loader := hpq.item
  hpq.remove
  -- Get next event
  l_item := l_loader.item
  if l_item.time < min then
    -- Go to next event
    l_loader.forth
    if not l_loader.after then
      hpq.extend (l_loader)
    end
  elseif l_item.time > max then
    -- Don't do anything, don't add to the queue
    else
      -- Call progress action
      if l_current = Void or else l_current < l_item.time then
        l_current := l_item.time
        if progress_action /= Void then
          progress_action.call ([l_current])
        end
      end
      -- Visit item
      l_item.visit (a_visitor)
    end
  end
  -- Call progress action
  if progress_action /= Void then
    progress_action.call ([max])
  end
  -- Cleanup
  a_visitor.cleanup
end

A.4 Examples

Listing A.10: Eiffel: Class: Baboon Factory

class
  BABOON_FACTORY
create {BABOON CROSSING}
  make_with_rope
feature {NONE} -- Initialization
  make_with_rope (a_id: like id; a_rope: separate ROPE; a_num: like baboons_count)
    -- Creation procedure.
    do
      rope := a_rope
      id := a_id
end

baboons_count := a_num
end

feature {BABOON CROSSING} -- Basic Operations
live
--- Live.
local
i: INTEGER
t_baboon: separate BABOON
do
from
i := 0
until
i >= baboons_count
loop
create t_baboon.make_with_rope ((id * baboons_count) + i, rope)
launch_baboon (t_baboon)
--- Code for random sleep
i := i + 1
end
end

feature {NONE} -- Implementation
launch_baboon(a_baboon: separate BABOON)
--- Launch the baboon.
do
a_baboon.live
end
id, baboons_count: INTEGER
rope: separate ROPE
end
Appendix B

Glossary

**BON** Business Object Notation. It is a graphical notation for describing object-oriented programs and systems.

**C** C is a general-purpose computer programming language developed in 1972.

**CPU** Central Processing Unit.

**ECF** Extension of Eiffel Configuration File, based on XML.

**Eiffel** Object-oriented programming language designed by Bertrand Meyer.

**EIFGENs** Directory used by the Eiffel compiler to store temporary data. It is used by the profiler to store profile information and data.

**ETH** Eidgenössische Technische Hochschule, Swiss Federal Institute of Technology.

**EV** Short for EiffelVision, the Eiffel multiplatform graphics (user interface) library.

**EVE** Eiffel Verification Environment. It is the research branch of EiffelStudio developed at ETH.

**RAM** Random Access Memory.

**SCOOP** Simple Concurrent Object-Oriented Programming. It comes as a refinement of Eiffel to ease the development of concurrent applications.

**UNIX epoch** It is point in time, defined as midnight Coordinated Universal Time (UTC) of January 1, 1970.

List of Figures

2.1  Call flows  .......................................................... 11
3.1  Measurement points ................................................. 16
4.1  BON diagram of events .............................................. 24
4.2  Call flows with colored time intervals .......................... 24
4.3  BON diagram of visitors and loaders ............................. 26
4.4  BON diagram of profile abstraction .............................. 27
4.5  Structure of profile abstractions ................................. 31
5.1  Baboon Crossing profile 1 .......................................... 36
5.2  Baboon Crossing profile 2 .......................................... 36
5.3  Baboon Crossing execution times ................................. 40
5.4  Barber Shop profile 1 .............................................. 40
5.5  Barber Shop profile 2 .............................................. 41
5.6  Barber Shop execution times ....................................... 42
5.7  Graph of execution times for probe effect ....................... 44
7.1  Starting the profiler ............................................... 49
7.2  Profiler wizard: data type selection ............................ 49
7.3  Profiler wizard: directory selection ............................. 50
7.4  Profiler wizard: run selection ................................... 50
7.5  Profiler wizard: options .......................................... 51
7.6  Profiler wizard: visualization window .......................... 52
List of Tables

3.1 Comparison of file formats .................................. 17
3.2 Comparison of profiling hooks placement ..................... 19

4.1 Feature events collection places. .............................. 25
4.2 Classes involved in profile abstractions ...................... 32

5.1 Baboon Crossing execution times .............................. 39
5.2 Barber Shop execution times ................................. 41
5.3 Analysis of probe effect ................................. 43

7.1 Overview of ECF settings .................................. 48
# List of Listings

2.1 *Eiffel*: Example compilation: original code ........................................ 9  
2.2 *Eiffel*: Example compilation: proxy code .......................................... 10  
2.3 *Eiffel*: Example compilation: client code .......................................... 11  

4.1 *Eiffel*: Loader: continue ................................................................. 28  
4.2 *Eiffel*: Loader: delay ................................................................. 28  
4.3 *Eiffel*: Loader: resume ................................................................. 29  
4.4 *Eiffel*: Visitor: processor start ....................................................... 29  
4.5 *Eiffel*: Visitor: visit feature call ................................................... 30  
4.6 *Eiffel*: Visitor: external call .......................................................... 31  
4.7 *Eiffel*: Loading profile data into EV abstraction .................................. 32  
4.8 *Eiffel*: Applying a zoom action ....................................................... 33  

5.1 *Eiffel*: Original BABOON ................................................................. 37  
5.2 *Eiffel*: Modified BABOON ................................................................. 37  
5.3 *Eiffel*: Original ROPE ................................................................. 37  
5.4 *Eiffel*: Modified ROPE ................................................................. 37  
5.5 *Eiffel*: Original BABOON creation .................................................... 38  
5.6 *Eiffel*: Modified BABOON creation .................................................... 38  
5.7 *Eiffel*: Original Baboon Crossing ...................................................... 39  
5.8 *Eiffel*: Modified Baboon Crossing ...................................................... 39  

6.1 *Eiffel*: Profiler configuration options ............................................... 46  

A.1 *Eiffel*: Class information gathering ............................................... 58  
A.2 *Eiffel*: Feature information gathering ............................................... 58  
A.3 *Eiffel*: Collection of events by scheduler ......................................... 59  
A.4 *Eiffel*: Collection of events by proxy ............................................... 60  
A.5 *Eiffel*: Collection of wait condition events ....................................... 61  
A.6 *Eiffel*: Collection of processor events ............................................. 61  
A.7 *Eiffel*: Class: Loader ........................................................................ 62  
A.8 *Eiffel*: Class: Event visitor .................................................................. 63  
A.9 *Eiffel*: Loader: load ........................................................................... 65  
A.10 *Eiffel*: Class: Baboon Factory .......................................................... 66
Bibliography


