Chair of
Software Engineering

# Integrating SCOOP into EVE

Master Thesis

By: Patrick Huber

Supervised by: Benjamin Morandi
Prof. Dr. Bertrand Meyer

Student Number: 01-920-305

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

## Abstract

SCOOP (Simple Concurrent Object-Oriented Programming) is a simple but very powerful O-O contract based framework for concurrent programming. The idea which was initially published in 1990 has since been developed to a comprehensive model with enriched type system, generalized semantics of contracts, flexible locking policy and library implementation.

To use SCOOP one important step is missing: The integration in Eiffel – which defines the scope of the presented work. In this master thesis we integrate SCOOP into EVE – a research branch of EiffelStudio. The goal is to get SCOOP code compiled in EVE like any other Eiffel program. To achieve this goal, we change the parser, perform a AST to AST transformation and integrate the whole SCOOP compilation into the existing EVE compiler.

## Kurzfassung

SCOOP ist ein einfaches aber sehr mächtiges, objektorientiertes, Contracts basierendes Framework zur Realisierung von Nebenläufigkeit in Programmen. Aus der erstmals im Jahre 1990 vorgestellten Idee entwickelte sich ein umfassendes Modell, welches ein angereichertes Typensystem, ein verallgemeinertes Verständnis von Contracts, ein flexibles Verfahren des Lockings und eine Bibiotheksimplementierung mit sich bringt.

Um SCOOP nutzen zu können fehlt noch ein wichtiger Schritt: Die Integration in Eiffel. Diese Lücke wollen wir mit dieser Masterarbeit schliessen, indem wir SCOOP in EVE – einer Forschungsversion von EiffelStudio – einbauen. Unser Ziel ist SCOOP Code wie normalen Eiffel Code in EVE kompilieren zu können. Um dieses Ziel zu erreichen werden wir Änderungen am Parser vornehmen, eine AST zu AST Transformation durchführen und die SCOOP Kompilierung in den vorhanden EVE Kompiler integrieren.

## Content

## 1. Introduction

SCOOP - Simple Concurrent Object-Oriented Programming - is a practical framework for the development of high-quality concurrent software which carries the advantages of object technology and Design by Contract to the concurrent context. Its simplicity relies on basic O-O concepts; its expressiveness and modelling power is due to the full support for advanced O-O mechanisms and Design by Contract. Unlike most existing concurrent O-O languages, SCOOP is a full-blown O-O language: it supports (multiple) inheritance, polymorphism, dynamic binding, genericity, and contracts. One main goal of SCOOP is to hide the synchronization problems - a major problem in the concurrent programming context - from the programmer.

## 2. Goal

The EVE project provides a research branch of EiffelStudio. The goal is to avoid individual modifications of EiffelStudio. EVE, code-named for ETH Verification Environment, includes the outgrowth of CDD, AutoTest, Ballet, Origo plug-in, Escher, Proof-Transforming Compilation plugin and anything else we may dream of in the future.

The goal of this master thesis is to integrate the SCOOP model into EVE. In the end it should be possible for developers to write, compile and run SCOOP programs in EVE.

## 3. Related work

The SCOOP story began in 1990 where Bertrand Meyer presented the idea of the SCOOP model the first time. An article contained an informal description where elements like separate objects, processors and wait-conditions were mentioned. A first refined version was published in 1993 in a second article which addresses primarily requirements like mutual exclusion and condition synchronization. In 1997 Bertrand Meyer published his Book OOSC2 [1] which contains a detailed description of the model including examples and some advanced mechanisms such as duels (priority scheduling) and CCF (manual processor specification).

### 3.1 Piotr Nienaltowski's dissertation

Piotr Nienaltowski took the concept of the SCOOP model presented in 1997 as starting point for his studies, compared it with other models, worked out its implication and filled the gaps by carrying out an in-depth analysis of the model, identified inconsistencies, extended and formalized the model and finally provided an implementation. He described his main results of his dissertation in [2] as follows:

- An enriched type system to detect and eliminate potential atomicity violations.
- A generalised semantics of contracts, applicable in concurrent and sequential contexts.
- A flexible locking policy to optimise the use of resources and to minimise the danger of deadlocks.
- A seamless integration of the concurrency model with Eiffel: a full-blown object-oriented language.
- A library implementation of SCOOP and a compiler which type-checks SCOOP code and translates it into pure Eiffel with embedded library calls; a supporting library of advanced concurrency mechanisms is also provided.

At this point the reader might ask what work is left to do for this master thesis – especially when there is already an implementation of the compiler. A first problem of the existing version was that its implementation is based on the GOBO 3.5 compiler – a compiler which generates a different abstract syntax tree compared to the current EVE compiler. Because of that, this SCOOP compiler was not at all portable to the current EVE. Furthermore, some constructs of the Eiffel language were not or not fully supported. Also the implementation was pretty hard to read and would have been even harder to maintain or extend – and was unfortunately not well documented.

Our solution Is based on the functionality of Piotr Nienaltowski's solution, but extends it in some considerable aspects.

## 4. Architecture

There are several ways to achieve the goal of integrating SCOOP into EVE, so that the IDE supports the compilation of SCOOP programs. This section describes the architectural approaches we considered. In conclusion we will pick the approach we favoured.

### 4.1 Intermediate AST approach

The main idea of the Intermediate AST approach can be described as follows: The parser recognizes the new syntax and creates a new SCOOP enriched AST (abstract syntax tree) when compiling user code. In a second step, a type checker verifies that the user code conforms to the SCOOP type system roles. In a third step, the enriched AST is transformed into two immediate “common Eiffel like” ASTs. Doing so, these intermediate ASTs will not contain any SCOOP specific type information any more, but rather rely on the SCOOP library. After this step the compilation can proceed as usual.

### 4.2 Implementation of the intermediate AST approach

When we started the project, the decision was already made to integrate SCOOP with the intermediate AST approach. But shortly after beginning, we recognized that this approach cannot be realized as it was intended. EVE is based on the physical classes saved in files. More precisely, for each class with a qualified name there is exactly one CLASS_C object in the system. Objects of this class are the basic instruments in EVE to handle the classes during the compilation steps. The AST created for a particular class is referenced from the corresponding CLASS_C instance. Furthermore a CLASS_C object can have several references to CLASS_I objects which represent the physical classes (text files) with the same name. To avoid ambiguity only one class file represented by the CLASS_I instances is compiled: This selected CLASS_I object – called ‘lace_class’ in CLASS_C – is either unique or overwriting the other CLASS_I objects with the same name. One of the descendants of the CLASS_I is the class EIFFEL_CLASS_I, which is defined by its attached physical source code file.

Regarding SCOOP and the intermediate AST approach, we want to generate for each original class two new classes based on its AST representation: First we create a new client AST class node with the same name as the original one has. Then we create a new proxy AST class node with the name 'SCOOP_SEPARATE__<name>'. To handle this new created proxy AST class node during the following compilation steps, we need to put it into a CLASS_C object with the new proxy class name. But as already mentioned this CLASS_C needs at least one physical representation, which we would not have.

The problem caused us to think about integrating SCOOP by creating the new client and proxy classes on a textual basis and save them in physical files to parse them in a second step. This change is not a conceptual change, but rather a detour. The AST of an intermediate AST to AST transformation would look exactly like the one we are generating by creating first new classes and reparsing them. Therefore there is no difference at the end - we just insert a processing step in between. In addition, this solution has the advantage that the whole transformation is much more readable in its implementation and also in its generated results.

## 4.3 Degrees: What to do where

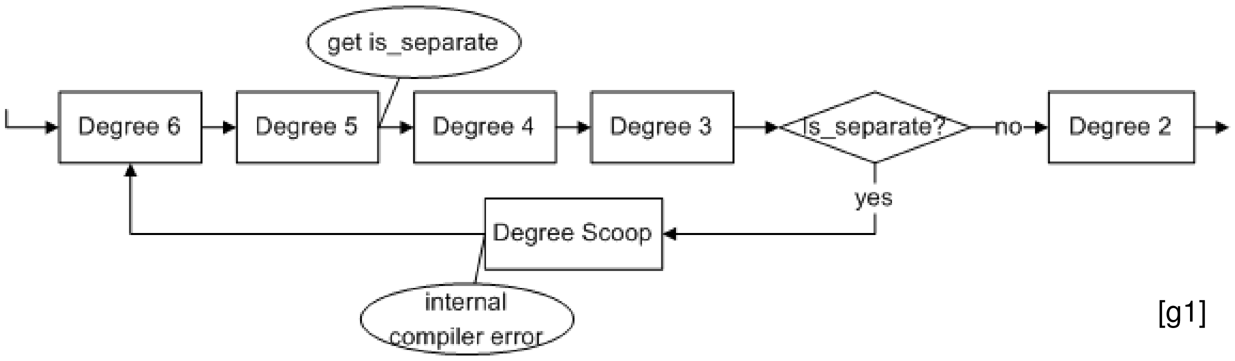
In this section we will give you a small overview on the Eiffel compiler before explaining in detail where currently the SCOOP compilation code is integrated.

The compilation in EVE happens during several steps called degrees. It starts with degree 6, where the configuration file is read and updated classes are collected. Degree 5 continues with getting new referenced classes and creating an abstract syntax tree while parsing them where necessary. In degree 4, the inheritance clause is analysed using a topological order and also feature tables are built, containing all available features of a class. Degree 3 validates the code of each routine and when it is valid, it generates a BYTE_NODE object representing a compiled version of the AST which is made for code generation purposes. In the next steps, degrees 2 and 1, melted code is generated. This melted code is used to generate C code in degree -1 for the workbench version. The degrees -2 and -3 produce finally optimized code, where techniques like inlining, dynamic to static binding optimization or removal of dead branches produce faster code.

The integration of the SCOOP compilation code happened in some refinements and affects the original Eiffel compiler only in some small points. This is especially interesting when compiling a project which not uses SCOOP code: In this case it is ensured that the SCOOP changes on the EVE compiler does not affect the compilation.

In degree 5 during the parsing progress, we get almost for free the information if any class has a 'separate' keyword in its code. If this is the case, we keep it in mind for the further processing steps. If there is no 'separate' keyword then EVE compiles as before.

We will now take a look at the compilation steps and how they are  implemented at this point. The first change is as already mentioned in degree 5. After parsing, we get the information about 'separate' keyword occurrences from the generated AST. It is assumed that there is SCOOP code recognized. After processing degree 5 and 4, we check the code according to the SCOOP type system roles in degree 3, the type checker. If the type checker doesn't complain, we get into a new degree called DEGREE_SCOOP, which produces for each updated or newly created class a proxy and a client version. DEGREE_SCOOP also creates a starter class and marks all created classes with an overwriting property. Finally we create an artificial internal error which lets the Eiffel compiler restart the whole compilation. This way, it simply starts again with degree 6, then degree 5 and so on. The difference between the second attempt to compile and the first is that the new created code is free of the SCOOP syntax and compiles now with the EVE compiler just like EVE used to do without our modifications.



[g1]

## 4.4 Discussion

As an alternative to the Intermediate AST approach a 'runtime approach' can be mentioned. This approach keeps the SCOOP information after type checking in degree 3 down to degree -1 where C code is generated. Therefore, the functionality of the SCOOP library needs to be integrated in the runtime. The advantages of this approach are obvious: It is faster at compile time because processing happens as part of the code generation. But it is also faster at runtime because calls in C code are faster than calls to an Eiffel library.

On the other hand, this approach is harder to implement and maintain as well as less flexible, because it requires knowledge of C code generation. This solution reduces further the portability to other operating systems because it involves C code generation.
As EVE is a research branch of EiffelStudio, flexibility is favoured over efficiency. Thus the intermediate AST approach is the approach of choice. Nevertheless efficiency must be a key requirement.

At this point the reader might ask the question “Is this way of integrating SCOOP efficient?” The answer is surprisingly very simple: Yes. The current solution – creating new classes, restarting the compilation and making a second turn – generates indeed some overhead. So why can we simply answer with Yes? EVE compiles only classes which are updated since the last compilation. Therefore only few classes have to be created and recompiled in the second turn – and this is almost bearable.

## 5. Code processing

SCOOP is supported by two tools on an implementational basis: The SCOOP library (SCOOPli) implements the basic computational model. It contains the processors, atomic locking, scheduling, wait by necessity and contract semantics. This tool was part of [2] and therefore already available. The second tool is a compiler which validates and transforms SCOOP syntax enriched source code into pure Eiffel code with embedded calls to SCOOPli. This compilation tool is not mandatory when using the SCOOPli, but manually wrapping separate calls, creating proxies and the explicit calls to the scheduler is complex and therefore very error prone. Therefore we provide a compiler to process this transformation.

### 5.1 Client and proxy classes

Processing a SCOOP class implies the generation of two new classes: a client class with name of the original class and a proxy class whose name gets the prefix 'SCOOP_SEPARATE__'.
In the client class, SCOOP constructs get replaced with appropriate calls to the SCOOP library. This is realized by implementing a redirection over the proxy class. There are also other modifications necessary – for example: Inheriting from SCOOP_SEPARATE__CLIENT, turning the declaration of separate types into the declaration of the corresponding proxy class, adding an additional argument when calling on separate objects and therefore also replacing infix notations, revising creation calls with respect to the processor and also revising assignment attempts. Further, every routine has to be split into three routines: one routine containing locking functionality, one routine for wrapping the original body and a last one for wrapping the wait conditions.
The proxy class on the other hand implements the separate type and provides support for redirected calls to SCOOPli taking into account mechanisms like lock passing. There is no inheritance relation between the client and the proxy class. Delegation is used to access features of the original class from a proxy class.
At this point we would like to refer to [2], chapter 11, if the reader is interested in more details.

## 5.2 Assigner solution

One of the bigger challenges while implementing the proxy classes was to find an appropriate solution for the assigner tag – which causes some problems. We will take a closer look at this construct.

During the generation of a proxy class SCOOP_SEPARATE__C out of an original class C, every query f in the original class leads to the generation of a query f in the proxy class. The signature of the query in the proxy class is based on the signature of the query in the original class. There are however two important differences. To begin with, every non-expanded type T of a formal argument in the original class gets replaced in the proxy class by the proxy version of T. As this processing rule is not relevant for the current discussion, we will not go into more details at this point. The second difference is more important for this context: The signature of f in the proxy class gets expanded with an additional formal parameter called a_caller_ of type SCOOP_SEPARATE_TYPE as a first formal argument.

In combination with assigner features, the additional formal parameter leads to a complication. Consider the following shortened pair of features in an original class C:

```
item (i: INTEGER): T assign put
  do
     …
  end

put (v: T; i: INTEGER)
  do
     …
  end
```

The command put is used as an assigner for the query item. As a benefit of this declaration, it is possible to assign a value to item, regardless of whether item is a function or an attribute. Without the assigner such an assignment would only be possible if item is an attribute and the assignment destination is unqualified. For the details on this see section 8.22.2 on the syntax of assignments in [3].

The shortened proxy code in the proxy class SCOOP_SEPARATE_C for the above features is shown below:

```
item (a_caller_: SCOOP_SEPARATE_TYPE; i: INTEGER): T assign put
  local
    …
  do
    …
  end


put (a_caller_: SCOOP_SEPARATE_TYPE; v: T; i: INTEGER)
  local
    …
  do
    …
  end
```

Note that the feature item must have an assigner in order to support assignment patterns, where the target in the assignment destination is separate. The above code however will not compile, because the validity rules for assigners in section 8.5.22 of [3] expect the first argument of the assigner feature put to represent the value to be set. This requirement clashes with our processing rule, where every proxy feature has a_caller_ as the first formal argument.

### Generalization

To be more general, we consider an original class C containing a query f with an assigner.

```
f(p1: OT1; … ; pn: OTn): OTR assign p
```

This feature assumes the immediate or inherited existence of a feature p in the original class.

```
p(v: OTR; p1: OT1; … ; pn: OTn)
```

According to our current processing rules, the original class feature f results in the following feature in the proxy class SCOOP_SEPARATE_C:

```
f(
  a_caller_: SCOOP_SEPARATE_TYPE;
  p1: PT1; … ; pn: PTn
): PTR assign p
```

The code assumes the immediate or inherited existence of a feature p in the proxy class.

```
p(
  a_caller_: SCOOP_SEPARATE_TYPE;
  v: PTR;
  p1: PT1; … ; pn: PTn
)
```

### Solution – basic case

We solve the issue above with an additional processing rule. For every feature f in an original class C such as the one above, we create an additional assigner mediator feature f_scoop_separate_assigner in the proxy class SCOOP_SEPARATE_C. Its role is to mediate between the proxy class assigner p and the proxy class query f. For this reason the assigner mediator feature must have the right signature in order to act as an assigner for the feature f in the proxy class. More precisely, the first argument of the assigner mediator feature needs to be of the same type as the result type of the feature f in the proxy class. The remaining arguments need to be the same as the ones from f in the proxy class.

```
f_scoop_separate_assigner(
  v: PTR;
  a_caller_: SCOOP_SEPARATE_TYPE;
  p1: PT1; … ; pn: PTn
)
```

The assigner mediator feature is deferred in case f is deferred in the original class. This rule ensures the consistency of the assigner mediator with its query in terms of effectiveness. In case f is effective, the assigner mediator must have a body that calls the actual assigner p in the proxy class with the right order of actual arguments.

```
  do
    p(a_caller_, v, p1, …, pn)
  end
```

The newly introduced assigner mediator must then be used as the assigner for the query f in the proxy class.

```
f(  a_caller_: SCOOP_SEPARATE_TYPE;
  p1: PT1; … ; pn: PTn
): PTR assign f_scoop_separate_assigner
```

Note that the feature p in the proxy class does not get altered.

```
p(
  a_caller_: SCOOP_SEPARATE_TYPE;
  v: PTR;
  p1: PT1; … ; pn: PTn
)
```

### *Solution – considering inheritance*

So far we did not mention inheritance. In fact it is necessary to make sure that the assigner mediator feature conforms to the validity rules even when it comes to inheritance. For example, there are issues when an original descendant class B leads to the generation of an assigner mediator f_scoop_separate_assigner in a proxy class SCOOP_SEPARATE_B and an original ancestor class A already triggered the generation of an assigner mediator f_scoop_separate_assigner in a proxy class SCOOP_SEPARATE_A. In such a case, the assigner mediator feature f_scoop_separate_assigner in SCOOP_SEPARATE_B needs to redefine the inherited assigner mediator feature. In general we need to satisfy the following validity rules and properties from [3]:

- validity of feature declarations, described in section 8.5.25
- deferred class property, described in section 8.10.23
- validity of feature redeclarations, described in section 8.10.26
- validity of feature joins, described in section 8.10.28

The general idea towards solving such issues is to follow the inheritance structure of every original class feature f with an assigner in order to use this structure on the assigner mediator features in the proxy classes. With this approach we want to avoid any validity rule violations by taking the inheritance structure of the original class feature f as a role model.

This can mostly be done by using the inheritance clauses of the original classes and apply them to the assigner mediator features if necessary. However, making a deferred feature effective, is implicit and thus this case cannot be detected based on the inheritance clauses. These general remarks lead to the following set of rules:

- For every rename clause of a feature h to a feature g in an original class B, we check whether there is a proper ancestral version of g with an assigner. In this case the proxy class SCOOP_SEPARATE_B contains a rename clause from h_scoop_separate_assigner to g_scoop_separate_assigner.
- According to the validity rules for assigner commands in section 8.5.22 in [3] we set the export state of the assigner mediator feature feature to NONE. The export clause has no effect on the assigner solution.
- For every undefined clause of a feature g in an original class B, we check whether there is a proper ancestral version of g with an assigner. In this case

the proxy class SCOOP_SEPARATE_B contains an undefine clause for g_scoop_separate_assigner.

- For every redefine clause of a feature g in an original class B, we check whether there is a proper ancestral version of g with an assigner. Additionally we check whether g in the original class has an assigner. If both conditions are true then the proxy class SCOOP_SEPARATE_B contains a redefine clause for g_scoop_separate_assigner. Note that the second condition is necessary, as there won't be an immediate assigner mediator feature for g in the proxy class SCOOP_SEPARATE_B, if g does not have an assigner in the original class B. Thus a redefinition of the assigner mediator would be wrong.
- For every select clause of a feature g in an original class B, we check whether there is a proper ancestral version of g with an assigner. In this case the proxy class SCOOP_SEPARATE_B contains a select clause for g_scoop_separate_assigner.
- For every effective query g in an original class B, where the direct ancestor of g is deferred, we check whether there is a proper ancestral version of g with an assigner. Additionally we check that g in the original class B has no assigner. If both conditions are true, we have to deal with an implicit effecting of a feature, which needs to be applied to the inherited assigner mediator feature. In such a case we introduce an empty, but effective assigner mediator g_scoop_separate_assigner in proxy class SCOOP_SEPARATE_B. This is however only necessary in case this it is not already triggered by the presence of an assigner of g in original class B.

All of these rules check whether there is a proper ancestral version with an assigner for a feature g. This condition is the trigger for the generation of an assigner mediator for the feature g in a proper ancestor. If the condition is not satisfied then there won’t be an inherited assigner mediator for the feature g and consequently it will not be necessary to apply inheritance clauses to such a non-existent inherited assigner mediator.

# 6. Implementation

## 6.1 Parser

The first task in order to get to a compiler supporting SCOOP code is to extend the syntax rules. This section discusses the changes on the parser and its related environment. But before going into detail, we will summarize the decisions taken regarding the syntax.

- No enriched type annotations: The extended SCOOP syntax permits besides the usual "<>" notation for processor tags also an alternative comment-based notation "--<>". It should allow the use of SCOOP within Eiffel editors which are not (yet) aware of the full SCOOP syntax. We decided to not support this enriched type annotation.
- The "handler"-keyword problem: As syntactically described below in more details, the explicit processor specification can either be qualified or unqualified. An unqualified explicit processor specification relies on a processor attribute p declared in its class or one of its ancestors where p is of type PROCESSOR. A qualified explicit processor specification has the form e.handler where e is an entity of attached type. The idea was to define the handler notation as a keyword. This leads unfortunately to inconsistencies with existing Eiffel code. So we decided to not consider handler as a keyword, but rather as a feature name.
- Generics & formal generics: Although there are some open questions regarding genericity (see section 9.2 in [2]), the syntax of actual generic parameters, as well as formal generic parameters, is fully supported.

### *Grammar*

According to the decisions described above and the description given in [2], we define the grammar for SCOOP in Eiffel with the following properties:

- The explicit processor specification tag should only be allowed when the separate keyword is used
- The explicit processor specification can be defined in two ways:
  - Unqualified: < p > where p is of type PROCESSOR
  - Qualified: < e.handler > where e is a non-writeable entity of attached type.

The type of an attribute could therefore look as follows:

```
type :=  [detachable_tag]
         ["separate" [explicit_processor_specification]] class
         [actual_generics]
```

with ...

```
detachable_tag := "!" | "?"
```

and ...

```
explicit_processor_specification :=
     "<" entity ".handler" ">" | "<" entity ">"
```

We introduced therefore two new tags in the grammar file, called Explicit_processor_specification and Processor as elements of the type EXPLICIT_PROCESSOR_SPECIFICATION_AS. This new type summarizes the information of the explicit processor specification when given in a class and symbolizes at the same time a new node in the AST.

```
%type <EXPLICIT_PROCESSOR_SPECIFICATION_AS>
     Explicit_processor_specification Processor
```

### *Parser Integration*

The integration of the explicit processor specification tag into the abstract syntax tree (AST) is realized by adding the corresponding creation instruction to the AST_FACTORY. Further, we enriched the current AST type nodes with the new information about the processor to complete the SCOOP extended type information: This affected the two classes:

- CLASS_TYPE_AS
- NAMED_TUPLE_TYPE_AS

The type information of generic classes is specified in the GENERIC_CLASS_TYPE_AS class but fortunately this class inherits from CLASS_TYPE_AS. Of course the visitor classes, like the AST_ROUNDTRIP_ITERATOR, have to be updated for all type classes according to the new structure.

### *Syntax Checking*

As described in the introductory part, we are checking the syntax of the explicit processor specification to be compliant with the definition and create a new syntax error if this is not the case.

## 6.2 Type Checker and Type Derivation Engine

As explained in section 4.21, checking the conformance of the input code with respect to the SCOOP typing rules is done in degree 3 before the new classes are generated. Unfortunately, we didn't get to the extension of the type checker due to time constraints. Nevertheless during the code processing we required a way to derive the SCOOP extended type information for expressions and alike. Ideally this information would be made available by the type checker. Unfortunately, the type checker generates primarily BYTE_NODE objects representing a compiled version of the AST which is made for code generation purposes. So we are forced to get the type information – where not already available – by interpreting the AST's with an own tool. This little type getting functionality is implemented in some visitors collecting and visiting type nodes or expressions and combined it with the class system of EVE for inheritance purposes.

## 6.3 Code processor

In this current part we will explain how the client and proxy classes are generated and note some interesting points, which kept us busy during the implementation.

### 6.31 Roundtrip parser

The basic functionality of the code processor is to generate client and proxy classes out of the  SCOOP enriched AST that got built in degree 5. Originally, the idea of an AST is to provide abstract information of the written source code without formatting information like indentation, letter case or comments. Not even keywords are represented as nodes in a normal AST. On the other hand, many applications such as ours, can benefit from the information which is now missing in the AST, but still want to base their processing on the AST. One way of providing support for these applications is to enrich the AST with additional information which is often realized by using a trailing break mechanism: Statements or comments are attached to the preceding terminal symbol. EVE implements this concept differently with its roundtrip parser: All tokens (comments, keywords and separators are considered to be tokens as well) are stored in a linked list. There exists for example a token for a comment in the linked list, but the token itself is not part of the AST. As a result we get an enriched AST with references to a linked list containing the original source code including formatting, comments and keywords, which can be flexibly used for refactoring purposes.

With the roundtrip parser it is now possible to traverse the AST and print specific linked list nodes that are relevant for the current AST node, such as comments, indentations and so on.

### 6.32 Code processing with visitors

Traversing an AST can be implemented with the help of the visitor pattern. This design pattern enables us to separate an algorithm performed on an object structure, such as an AST, from the structure itself. Every visitor defines a particular order on how to traverse the AST nodes. It is possible to skip some parts of the AST or to repeat processing other parts of the AST. In our approach we use visitors to print out source code based on the AST.

### 6.33 Context specific visitor

In some situations it is necessary to have context dependant processing of nodes. For example let's consider the situation where a type name stored in a type node must be printed in different ways, depending on where the type appeared in the surrounding original class. In one context it is necessary to have a prefix and in another context a prefix is not desired. In another situation we want to process a feature call in different ways, depending on where the feature call is found in the surrounding original class. In case the feature call is part of a feature body, we want to pass on the processed formal arguments of the surrounding feature. In another context this is not desirable. We considered a number of ways to handle situations such as the ones described above. In the following we will go through a number of possibilities  and assess them.

### 6.34 Considering implementation alternatives in visitors

A first solution simply processes the nodes below the current node in the current node itself to ensure the context awareness. For example when processing a feature, we also process its result type by potentially adding a prefix, instead of visiting the type node with the visitor. Some negative effects of this solution are that we are bypassing the visitor structure and that we are duplicating code instead of fostering reusability.
A second possible alternative is the usage of flags to remember the context. For example we introduce a new flag which denotes whether we are currently processing a type with a prefix or without. A subsequent call to the process feature of a type node will take this flag into account. This approach supports the visitor structure and allows for a very easy way to keep information for processing steps further down in the AST structure. The drawback of this approach is that the code might become unreadable

when we have more than a few flags. Such a solution will quickly become very difficult to maintain, as it will be hard to keep track of flag values and their relevancy in a particular visitor feature.

As a third possibility we can use a hashtable to map context to a particular type of node. If a process feature for a node wants to pass on a context  to a process feature for node b, then node a creates an entry in the hashtable with the name of node b as the key and the context as the value. The process feature for node b can then access the context by querying the hashtable using the name of node b as the key. This solution solves the problem with a simple solution and an easy design. Unfortunately, this solution is not sophisticated enough to have an easy way of passing on a context to visitor features of a whole group of nodes. Furthermore it is difficult to expand the context if multiple visitor features want to set the context for a visitor feature of a single node.

Last but not least, it is possible to use different visitors for distinct parts of the AST. For each part that is thematically or intentionally distinct from other parts, we create a new visitor which processes the corresponding nodes in this particular context. To avoid duplicated code we suggest to inherit from a basic visitor which implements functionality to be used by every context specific visitor. Drawbacks of this approach are the complex design of the whole processing steps, especially if at the same time the code should be easy to read. This approach implies also to create many different visitors which should properly interact when parametrizing nodes. Nevertheless, the advantages of this approach are obvious: Based on a clear design you get a readable and maintainable solution which solves the problems around context awareness while processing nodes.

### Our choice

Our solution uses context specific visitors, because of their natural support for the thematically dependant processing of nodes. Therefore e.g. the processing of types, parents, features and assertions is implemented in context specific visitors. To avoid unnecessary code duplication, most of the visitors inherit from other common visitors. E.g. the visitors processing the parents of the proxy and the client classes inherit from a common parent visitor. Extremely helpful in this context was the precursor statement which allows to reuse functionality of a parent while having the possibility to specialize it in the current visitor. On the other hand, there are also situations, where code duplication with this approach is hard or simply not possible to avoid. For example, consider the situation shown in the following listing. Class A has a feature process_node_a where we process a fist sub node, collect some data and then process a second sub node. If we now inherit from this first visitor and redefine the process feature, because we have to do something new between the calls to the subnode process features, then we have no possibility to reuse the old feature with a precursor statement. The only way to avoid such situations is to keep visitor inheritance structures as simple as possible.

```
class A

...

process_node_a (l_as: NODE_AS)
is
  do
     process (l_as.subnode1)
     collect_data(l_as)
     process (l_as.subnode2)
  end
```

```
class B

inherit
  A
     redefine process_node_a end

process_node_a (l_as: NODE_AS)
is
  do
     process (l_as.subnode1)
     collect_data(l_as)
     do_somehting(l_as)
     process (l_as.subnode2)
  end
```

In some distinct  instances we made use of the flag approach, where such flags were well defined in terms of their meaning and their usage. We picked this approach in case we wanted to provide some context information to later invocations of process features but we could not justify a new context specific visitor. For example, there are situations where we want to print the result type of a feature with a prefix in cases where the result type itself is separate. To do so we use a type visitor where we get the separate status of the type node. To print out the class name of the type we have to call the process feature of the type’s direct subnode called ID_AS – as we don’t want to bypass the visitor structure. Before the desired name is printed, the process feature of the id node first processes some unprocessed list entries like spacing. This means that we have to print the prefix in the process feature of the id node. Therefore we should know at this point the separate status of the processed result type. Here, the usage of a flag is absolutely meaningful, since we know that in the context of this type visitor no other node calls the process feature of the id element and creating a new visitor for visiting the id node makes no sense.

### 6.35 Visitor structure - an overview

In EVE the AST_VISITOR class represents general visitor objects. The class AST_ROUNDTRIP_ITERATOR inherits from AST_VISITOR and represents visitors that iterate over the AST with respect to the underlying linked list. Our code processors are based on this very AST_ROUNDTRIP_ITERATOR class. Diagram [g2] gives an overview over the visitors used in the code processing step. All of the displayed classes inherit directly or via SCOOP_AST_CONTEXT_PRINTER from AST_ROUNDTRIP_ITERATOR.

### *SCOOP_CONTEXT_AST_PRINTER*

The SCOOP_CONTEXT_AST_PRINTER provides basic printing functionality based on the visitor structure which gets revised and refined by the descendants for special purposes. As you might have noticed most of the classes inherit from SCOOP_CONTEXT_AST_PRINTER. The reason is to give all of them the possibility to print their generated code directly to a single context, the ROUNDTRIP_CONTEXT, while processing the nodes.

### *SCOOP_SEPARATE_PROXY_PRINTER, SCOOP_SEPARATE_CLIENT_PRINTER*

The two classes SCOOP_SEPARATE_PROXY_PRINTER and SCOOP_SEPARATE_CLIENT_PRINTER serves as a starting point for the code processing of the client and the proxy class. They process the body of the class by relying on context specific visitors such as the SCOOP_*_PARENT_VISITOR or the SCOOP_*_FEATURE_VISITOR to process the corresponding nodes. Finally these prime visitors return the processed code.

### *SCOOP_CLIENT_PARENT_VISITOR, SCOOP_PROXY_PARENT_VISITOR*

The two classes SCOOP_CLIENT_PARENT_VISITOR and SCOOP_PROXY_PARENT_VISITOR process the conforming and the non-conforming parents of an original class. Both inherit from the SCOOP_PARENT_VISITOR class which provides basic parent processing. The class SCOOP_PARENT_VISITOR itself inherits from the SCOOP_CONTEXT_AST_PRINTER class. The class SCOOP_PROXY_PARENT_VISITOR implements the assigner solution by analyzing the inheritance clauses of the original class.

### *SCOOP_PROXY_FEATURE_VISITOR, SCOOP_CLIENT_FEATURE_VISITOR*

The two classes SCOOP_CLIENT_FEATURE_VISITOR and SCOOP_PROXY_FEATURE_VISITOR implement the processing of feature nodes and print them to the current ROUNDTRIP_CONTEXT. The SCOOP_CLIENT_FEATURE_VISITOR implements in contrast to its proxy version the creation of three different client class features out of a single original class feature: the locking request feature, wait condition wrapper and enclosing routine feature. This is done by using another set of visitors (SCOOP_CLIENT_FEATURE_*_VISITOR).

### *SCOOP_CLIENT_FEATURE_ASSERTION_VISITOR*

As already described, original routines are split into three new client routines. While the second one just encloses the original routine, the other two routines implements locking and wait condition functionalities. The generation of the later two depends on the pre- and postcondition of the original routine. To support this, the SCOOP_CLIENT_FEATURE_ASSERTION_VISITOR queries and preprocesses the assertions with help of the SCOOP_CLIENT_ASSERTION_EXPR_VISITOR to make the information directly accessible.

### *SCOOP_TYPE_VISITOR, SCOOP_PROXY_TYPE_VISITOR*

The SCOOP_TYPE_VISITOR and the SCOOP_PROXY_TYPE_VISITOR return – as you might expect by reading the class name – type information about a visited type node. The information is in most cases directly accessible by visiting the node. Other information like 'is the class of deferred type' has to be looked up in the system or must be evaluated by querying the associated class. The reason why we have a second type checker for the proxy class is simple: Processing the type follows different processing rules, depending on whether we are generating a client class or a proxy class.

### *SCOOP_GENERICS_VISITOR*

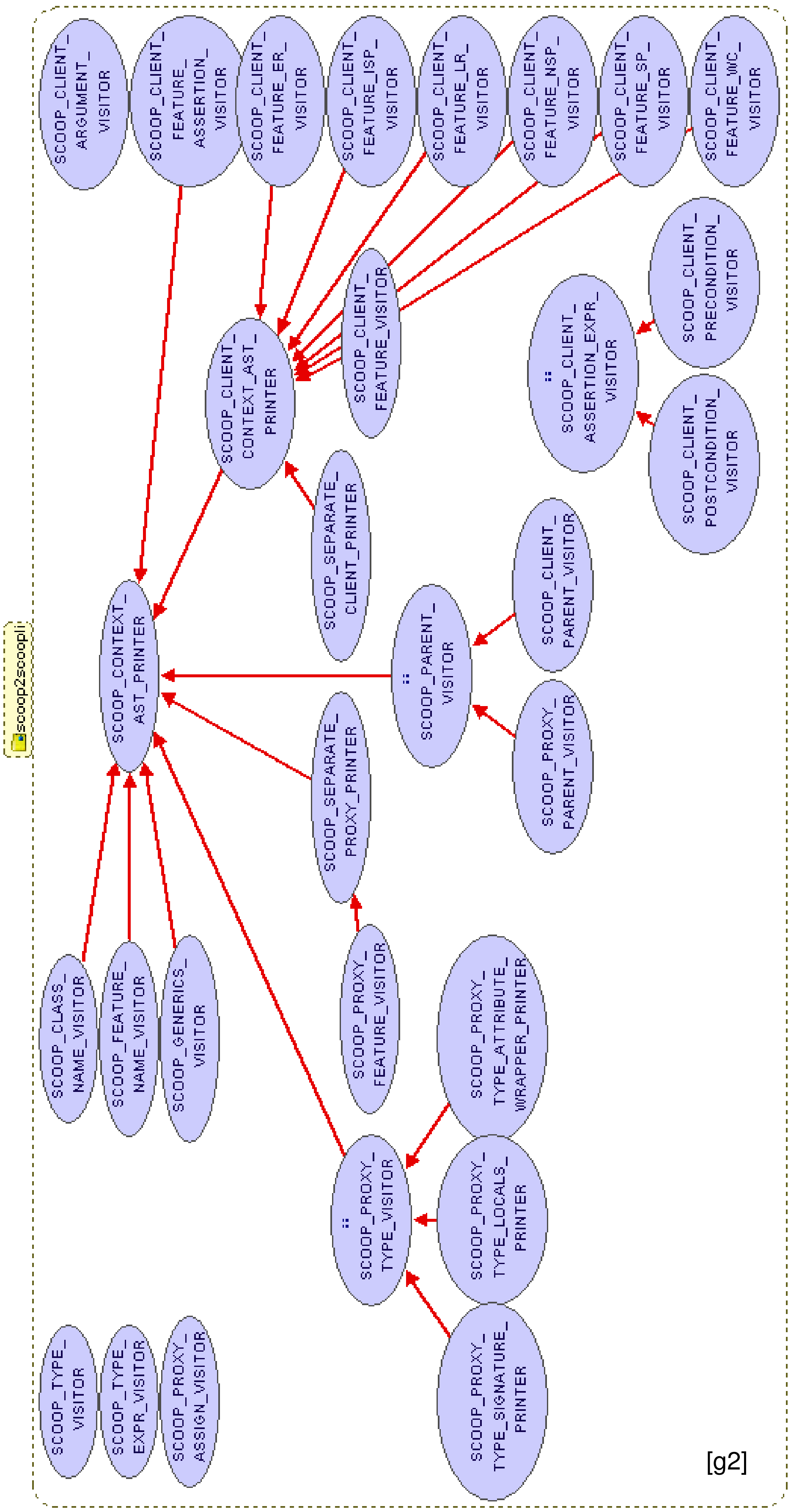
The SCOOP_GENERICS_VISITOR class inherits from the SCOOP_CONTEXT_AST_PRINTER class and processes the generic version of a type node. To build an own visitor for generic types was necessary since they might appear in a nested manner which requires recursive processing of subnodes.

### *SCOOP_CLASS_NAME_VISITOR, SCOOP_FEATURE_NAME_VISITOR*

The SCOOP_CLASS_NAME_VISITOR class and the SCOOP_FEATURE_NAME_VISITOR class both inherit from the SCOOP_CONTEXT_AST_PRINTER class. The SCOOP_CLASS_NAME_VISITOR processes a given class name node and prints directly to the ROUNDTRIP_CONTEXT. In contrast, the SCOOP_FEATURE_NAME_VISITOR only queries a feature name node and returns its name.

### *SCOOP_PROXY_ASSIGN_VISITOR*

The SCOOP_PROXY_ASSIGN_VISITOR class inherits directly from the AST_ROUNDTRIP_ITERATOR and implements together with the SCOOP_PROXY_ASSIGN_FINDER basic query functionality which is used for the described assigner solution.


scoop2scoopli
SCOOP_CLIENT_ARGUMENT_VISITOR
SCOOP_CLIENT_FEATURE_ASSERTION_VISITOR
SCOOP_CLIENT_FEATURE_ER_VISITOR
SCOOP_CLIENT_FEATURE_ISP_VISITOR
SCOOP_CLIENT_FEATURE_LR_VISITOR
SCOOP_CLIENT_FEATURE_NSP_VISITOR
SCOOP_CLIENT_FEATURE_SP_VISITOR
SCOOP_CLIENT_FEATURE_WC_VISITOR
SCOOP_CLIENT_CONTEXT_AST_PRINTER
SCOOP_CLIENT_FEATURE_VISITOR
SCOOP_CLIENT_PRECONDITION_VISITOR
* SCOOP_CLIENT_ASSERTION_EXPR_VISITOR
SCOOP_CLIENT_POSTCONDITION_VISITOR
SCOOP_SEPARATE_CLIENT_PRINTER
SCOOP_CONTEXT_AST_PRINTER
* SCOOP_PARENT_VISITOR
SCOOP_CLIENT_PARENT_VISITOR
SCOOP_PROXY_PARENT_VISITOR
SCOOP_SEPARATE_PROXY_PRINTER
SCOOP_CLASS_NAME_VISITOR
SCOOP_FEATURE_NAME_VISITOR
SCOOP_GENERICS_VISITOR
SCOOP_PROXY_FEATURE_VISITOR
SCOOP_PROXY_TYPE_ATTRIBUTE_WRAPPER_PRINTER
* SCOOP_PROXY_TYPE_VISITOR
SCOOP_PROXY_TYPE_LOCALS_PRINTER
SCOOP_PROXY_TYPE_SIGNATURE_PRINTER
SCOOP_TYPE_VISITOR
SCOOP_TYPE_EXPR_VISITOR
SCOOP_PROXY_ASSIGN_VISITOR
[g2]

## 7. Conclusion

The main results of this master thesis are:

- The prime implementation ideas found in Piotr Nienaltowski's dissertation are kept. We extended and refined the implementation concepts with missing elements.
- The parser knows and understands SCOOP syntax.
- We built a code processor which transforms the AST of SCOOP enriched source code into a pure Eiffel AST.
- Code processing is based on a context specific visitor structure.
- The overall design of the compiler is clear and readable, despite its complexity. Therefore it should be easy to maintain and extend.
- The implementation contains well known techniques and pattern like the factory pattern for the creation of the visitors.
- The integration into EVE is done by reusing the existing compiler instead of writing one from scratch.

### *Development*

The project webpage can found at [10] – the code is available at [11]

## 8. Future work

As future work, we suggest to …

- build a SCOOP type checker in a way which allows accessing type information of types and expression while creating client and proxy classes. This point would simplify the code processing and avoid a second separate and time consuming analysis of the types during the code processing step.
- implement a down- and upcast solution. Currently, assignment attempts and objects tests are not supported for separate objects.
- migrate from EVE 6.3 to EVE 6.4. For this its necessary to integrate 'void safety' in the code as well as in the SCOOP model.

Furthermore, the work on integrating SCOOP into EVE can be continued with …

- Editor: Implement automatic completion of the processor tag.
- Context tool (metrics): We might want to add a metric on the number of separate types.
- Documentation: Update required settings, restrictions, mixing with multithreading, etc.
- Navigation tool (cluster, feature): Change the cluster view in a way to hide overwrite classes, but to show client and proxy classes in combination with their original classes to display them when desired.
- Interface to external code: How will the new construct appear via external features, CECIL, .NET, external features?
- Integration into EiffelWeasel.

## 9. References and background material

[1] Bertrand Meyer: Object-Oriented Software Construction, 2nd edition, Prentice Hall, 1997.

[2] Piotr Nienaltowski: Practical framework for contract-based concurrent object-oriented programming.

[3] Standard ECMA-367, Eiffel: Analysis, Design and Programming Language, 2nd Edition, Ecma International, Rue du Rhône 114 CH-1204 Geneva, www.ecma-international.org, June 2006.

[4] Maurice Herlihy, Nir Shavit: The Art of Multiprocessor Programming. Morgan Kaufmann, 2008.

[5] Allen B. Downey: The Little Book of Semaphores Second Edition. Green Tea Press, 2005.

[6] Piotr Nienaltowski: Flexible locking in SCOOP. CORDIE'06, July 2006, York, UK.

[7] Piotr Nienaltowski, Bertrand Meyer: Contracts for concurrency, CORDIE'06, July 2006, York, UK.

[8] Nienaltowski P.: Refined access control policy for SCOOP, Technical Report tr511, ETH Zurich, February 2006

[9] Nienaltowski P.: Efficient data race and deadlock prevention in concurrent object-oriented programs, Doctoral Symposium, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Vancouver, Canada, October 2004. Appears in OOPSLA 2004 Companion: 56-57.

[10] SCOOP Origo project website: http://scoop.origo.ethz.ch/wiki/development

[11] SCOOP SVN project repository: https://svn.origo.ethz.ch/eiffelstudio/branches/eth/scoop/Src/