Master Thesis

Implementation of advanced SCOOP aspects

Author(s):
Müllhaupt, Damien

Publication Date:
2010

Permanent Link:

https://doi.org/10.3929/ethz-a-006050505 →

Chair of Software Engineering

Implementation of advanced SCOOP aspects

Master Thesis

Damien Müllhaupt
ETH Zürich
damienm@student.ethz.ch

October 26, 2009 - April 26, 2010

Supervised by:
Benjamin Morandi
Prof. Dr. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Advanced SCOOP[1] aspects play an important role in SCOOP projects. To support a full fledged SCOOP implementation we have to make sure that advanced SCOOP aspects are processed and translated correctly. The translation is done from SCOOP code which is processed and transformed into Eiffel code by running it through the SCOOP compiler. In order for SCOOP to be able and understand those advanced aspects we have to enrich the SCOOP compiler to know how to deal with those aspects while processing the SCOOP code.

Modern SCOOP projects can include feature redeclaration involving separate and non-separate types. Cases of feature redeclaration including separate and non-separate types which are allowed under the SCOOP conformance rules fail being processed by the current SCOOP implementation[2]. The solution is to simulate that no redeclaration was done on the feature signature while adapting the internal use of the changed type in the Eiffel code which was processed and translated from the SCOOP code.

As well as feature redeclaration, casting is an important advanced aspect of a language. Casts including separate and non-separate types are not fully supported in the current implementation. In order to support them, a conversion mechanism is inserted in the processed Eiffel code to deal with conversions and the translation of object tests, and assignment attempts are re-implemented.

Creating objects is a vital functionality in Eiffel as well as in SCOOP where we need to make sure that the creation of objects is processed and translated correctly. Creation instructions and creation expressions, create new objects in the SCOOP environment requiring a processor handling them in order to conform to the SCOOP model. This is not supported in the current SCOOP implementation. A correct translation of creation instruction and creation expressions is implemented involving processors which are assigned to the objects according to their nature.

Processors handling agent objects are not correctly assigned in the current SCOOP implementation where agent objects are always handled by the processor of their caller. The translation of the agent definition is implemented in the current SCOOP implementation but was not finished to a compiling state because the compiler makes assumptions on Eiffel base classes that we cannot accommodate at the moment.

[1] Simple concurrent object oriented programming
[2] Eiffel Verification Environment with SCOOP integration

Acknowledgments

I would like to thank Benjamin Morandi and Prof. Dr. Bertrand Meyer for supervising my Master Thesis. While Benjamin Morandi was the best possible tutor during my thesis, I very much appreciated the theoretical discussions with Prof. Dr. Bertrand Meyer. It was a pleasure to work in this team!

I wish to thank Prof. Dr. Bertrand Meyer for giving me the opportunity to work in his group. All the members of the Meyer group provided me a very cooperative working environment and helped me whenever I needed support or advice.

Contents

Chapter 1

Introduction

Designing and developing concurrent programs play a prominent role in today's software development. Concurrent programming is hard to implement correctly due to its complex nature, demanding close attention by the programmer. SCOOP offers a simple object oriented model on how to deal with concurrency. It relies on basic object oriented concepts to establish the simplicity of the language making it intuitive to use. The current version of the SCOOP implementation in the Eiffel Verification Environment[1] is a rework of the standalone implementation Piotr Nienaltowski implemented in his doctoral thesis[1]. This re-implementation was done in the master thesis written by Patrick Huber[2]. Due to time constraints some advanced aspects of the language were only partially implemented, leaving the need to expand and revise the current implementation to include missing advanced aspects. To work with SCOOP in EVE productively we have the need for a complete implementation of the language. In order to achieve that we have to master the integration and implementation of the advanced SCOOP aspects which were not fully supported yet. This master thesis covers and implements some of the advanced SCOOP aspects which are missing in the current SCOOP implementation, presenting the problems of each aspect as well as a solution including the implementation into EVE.

1.1 Related Work

In the master thesis of Patrick Huber[2] the current SCOOP implementation was described as the following: SCOOP compiles the code in the current implementation by spawning two classes from the original SCOOP class on compile time. The creation of the two classes is invoked in the feature *execute_on_class* of the class *DEGREE_SCOOP*. The first generated class which is called *Client* class, is the non-separate spawn of the SCOOP classes and includes the Eiffel code which was processed from the SCOOP code. It is generated by the client printer (*SCOOP_SEPARATE_CLIENT_PRINTER*) which processes the

[1] Research branch from EiffelStudio also called EVE

SCOOP class by calling the feature *process_class_as*. The second generated class which is called *Proxy* class, is the separate version and implements all the locking mechanisms needed in order to work with separate objects. The *Proxy* class is also spawned in the feature *execute_on_class* and is generated in the proxy printer (*SCOOP_SEPARATE_PROXY_PRINTER*). The proxy printer calls the feature *process_class_as* in order to invoke the processing of the SCOOP class and the generation of the *Proxy* class. In general both visitors, client printer as well as the proxy printer, have a sub-visitor in order to process features of the SCOOP class: The client feature visitor (*SCOOP_CLIENT_FEATURE_VISITOR*) which translates SCOOP features for the *Client* class and the proxy feature visitor doing the same for the *Proxy* class. When all the SCOOP classes are processed and broken down into the *Client* and the *Proxy* version, the code consists of Eiffel code and the compilation is restarted compiling an Eiffel project.

Chapter 2

Conformance and casts

2.1 Problem

Casts in SCOOP are different from casts in Eiffel because objects in SCOOP are defined also by the processor handling the object as well as their separate or non-separate nature. Regular conformance is not supported by the current[1] SCOOP implementation which is shown in snippet 2.1 where we try to assign a non-separate object to a separate object (a := b) which should be allowed according to the SCOOP conformance rules. Casts in form of object tests and assignment attempts shown in the code snippet 2.1 are not supported either, the current SCOOP implementation would have processed the shown snippet into code shown in snippet 2.2. Both of the object tests would have failed in that case since SCOOP_SEPARATE__B would have to be subtype of A, and B would have to be subtype SCOOP_SEPARATE__A in order for this to work. To be able to cast from separate to non-separate objects we need a mechanism in place which can convert client objects to their corresponding proxy objects on demand. Object tests and assignment attempts involving SCOOP objects demand for a more specific translation since SCOOP objects are also compared by the processor which handles them in comparison to normal Eiffel objects. This leads to a more generalized interpretation of the object test and the assignment attempt reflecting processors involved.

2.2 Solution

To compile the code in listing 2.1 in SCOOP we need to adapt the current SCOOP implementation to assure that the following properties are allowed:

- Conversion from the client object to the proxy object is possible and supported

[1] Eiffel Verification Environment with SCOOP integration

Listing 2.1: *Eiffel*: Conformance

```
1  class
2    APPLICATION
3  feature
4    cast is
5      local
6        a: separate A
7        b: B
8      do
9        create b
10       a := b
11       if attached {separate B} a then
12         -- Cast from separate to separate
13       end
14       if attached {B} a then
15         -- Cast from separate to non-separate
16       end
17     end
18 end
19
20 class
21   A
22 end
23 class
24   B
25 inherit
26   A
27 end
```

Listing 2.2: *Eiffel*: Conformance

```
1  class
2    APPLICATION
3  feature
4    cast is
5      local
6        a: SCOOP_SEPARATE__A
7        b: B
8      do
9        create b
10       a := b
11       if attached {SCOOP_SEPARATE__B} a then
12         -- Cast from separate to separate
13       end
14       if attached {B} a then
15         -- Cast from separate to non-separate
16       end
17     end
18 end
19
20 class
21   A
22 end
23 class
24   B
25 inherit
26   A
27 end
```

- Object tests and assignment attempts have a correct translation into Eiffel code

2.3 Conversion

Eiffel offers an excellent mechanism in order to allow conversion from clients to their proxy types which is denoted by the *convert* keyword in the class definition. The conversion command implemented in Eiffel allows insertion of an own conversion command named *proxy_* which returns the proxy object of the current client, on demand, if a conversion is needed. The conversion command consists of two parts on the client side, the actual feature definition of the *proxy_* command and the insertion of the command in the *convert* clause where the export status of the command is defined. The conversion command has to be exported to all targets of a possible conversion of the client class in order to allow casting as well as *SCOOP_SEPARATE__ANY* and *SCOOP_SEPARATE_PROXY*.

2.3.1 Implementation

The insertion of the conversion command definition into each class is triggered in the features *insert_conversion* and *add_proxy_feature* of the class *SCOOP_SEPARATE_CLIENT_PRINTER*. The feature *insert_conversion* places *proxy_* in the *convert* clause of the current class and computes all ancestors by calling *compute_ancestors_names* from the class *SCOOP_CLIENT_PARENT_VISITOR* to define the export status of the conversion command. *add_proxy_feature* implements the insertion of the actual feature definition of *proxy_* which creates a proxy object handled by the same processor as the client object and returns it.

A deferred version of the *proxy_* feature is inserted into the class *SCOOP_SEPARATE_CLIENT*. All SCOOP processed classes inherit from the deferred version of *proxy_* in *SCOOP_SEPARATE_CLIENT* which requires a redefinition clause for *proxy_* in every processed class. The redefinition clause is inserted by the feature *process_internal_conforming_parents* of the class *SCOOP_CLIENT_PARENT_VISITOR*.

If the current processed class is deferred the insertion of the conversion command is skipped because of the fact that there is never going to be an instance of a deferred class.

2.4 Object tests and assignment attempts

Object tests of the form of *attached* {*T*} *exp [as l]* are translated according to listing 2.3 in order to conform to the SCOOP rules. This translation deals with the processors attached to SCOOP objects as well as the client and proxy objects of the involved types in order to conform to the SCOOP rules.

Assignment attempts of the format *l ?= exp* have to be translated in a similar way as object tests. The translation of assignment attempts is shown in

Listing 2.3: *Eiffel*: Object tests

```
1  -- T: (!, non sep, C)
2        -- exp: (de, non sep, D)
3              attached {T} exp [as l]
4        -- exp: (de, sep, D)
5              exp /= Void and then exp.processor_ = processor_
                   and then attached {T} exp.implementation_ [
                  as l]
6  -- T: (!, <e.handler>, C)
7        -- exp: (de, non sep, D)
8              exp /= Void and then exp.processor_ = e.
                  processor_ and then attached {T} exp.proxy_ [
                  as l]
9        -- exp: (de, sep, D)
10             exp /= Void and then exp.processor_ = e.
                  processor_ and then attached {T} exp [as l]
11 -- T: (!, <p>, C)
12       -- exp: (de, non sep, D)
13             exp /= Void and then exp.processor_ = p and then
                   attached {T} exp.proxy_ [as l]
14       -- exp: (de, sep, D)
15             exp /= Void and then exp.processor_ = p and then
                   attached {T} exp [as l]
16 -- T: (!, T, C)
17       -- exp: (de, non sep, D)
18             exp /= Void and then attached {T} exp.proxy_ [as
                   l]
19       -- exp: (de, sep, D)
20             attached {T} exp [as l]
```

listing 2.4 which includes, similar to object tests, the processors assigned to the objects as well as client and proxy objects of the involved types.

The motivation of those translation is the same in both cases, object tests and assignment attempts. Both consist of a target and a source which are defined as 'target ?= source' in the case of an assignment attempt and as 'attached target source [as l]' in the case of an object test. The following cases occur with object tests and assignment attempts depending on the nature of the target and the source. A wild card (*) was inserted for the attachment symbol, it may vary depending on if it is an object test or an assignment attempt, but does not have any influence on the translation:

- *Target (*, non sep, ANY); Source (*, non sep, ANY)*

  The normal case where both objects are non-separate and nothing has to be done since this is covered by the Eiffel implementation.

- *Target (*, non sep, ANY); Source (*, sep, ANY)*

  The target is non-separate and the source is separate: In order to be able to cast the source to the target we have to make sure that the source is handled by the current processor which is handling the target. If that is the case we can make the actual cast on the client object of the source.

- *Target (*, <e.handler>, ANY); Source (*, non sep, ANY)*

  The target is separate with a qualified explicit processor specification and the source is non-separate: In order to be able to cast the source to the target we have to make sure that the source is handled by the processor of the qualified explicit processor specification of the target. If that is the case we can make the actual cast on the proxy object of the source.

- *Target (*, <e.handler>, ANY); Source (*, sep, ANY)*

  The target is separate with a qualified explicit processor specification and the source is separate: In order to be able to cast the source to the target we have to make sure that the source is handled by the processor of the qualified explicit processor specification of the target. If that is the case we can make the actual cast.

- *Target (*, <p>, ANY); Source (*, non sep, ANY)*

  The target is separate with an unqualified explicit processor specification and the source is non-separate: In order to be able to cast the source to the target we have to make sure that the source is handled by the processor specified in the unqualified explicit processor specification of the target. If that is the case we can make the actual cast on the proxy object of the source.

- *Target (*, <p>, ANY); Source (*, sep, ANY)*

  The target is separate with an unqualified explicit processor specification

Listing 2.4: *Eiffel*: Assignment attempts

```
1  -- l: (dl, non sep, C)
2        -- exp: (de, non sep, D)
3              l ?= exp
4        -- exp: (de, sep, D)
5              if exp /= Void and then exp.processor_ =
                   processor_ then
6                    l ?= exp.implementation_
7              else
8                    l := Void
9              end
10 -- l: (dl, <e.handler>, C)
11       -- exp: (de, non sep, D)
12             if exp /= Void and then exp.processor_ = e.
                   processor_ then
13                   l ?= exp.proxy_
14             else
15                   l := Void
16             end
17       -- exp: (de, sep, D)
18             if exp /= Void and then exp.processor_ = e.
                   processor_ then
19                   l ?= exp
20             else
21                   l := Void
22             end
23 -- l: (dl, <p>, C)
24       -- exp: (de, non D)
25             if exp /= Void and then exp.processor_ = p then
26                   l ?= exp.proxy_
27             else
28                   l := Void
29             end
30       -- exp: (de, sep, D)
31             if exp /= Void and then exp.processor_ = p then
32                   l ?= exp
33             else
34                   l := Void
35             end
36 -- l: (dl, T, C)
37       -- exp: (de, non sep, D)
38             if exp /= Void then
39                   l ?= exp.proxy_
40             else
41                   l := Void
42             end
43       -- exp: (de, sep, D)
44             l ?= exp
```

and the source is separate: In order to be able to cast the source to the target we have to make sure that the source is handled by the processor specified in the unqualified explicit processor specification of the target. If that is the case we can make the actual cast.

- *Target (*, T, ANY); Source (*, non sep, ANY)*

  The target is separate and the source is non-separate: We can make the actual cast on the proxy object of the source if it is non *void*.

- *Target (*, T, ANY); Source (*, sep, ANY)*

  The target is separate and the source is separate: Since both types conform to each other nothing has to be translated.

2.4.1 Implementation

In the case of object tests we translate a boolean expression into a more complex boolean expression which allows a direct translation with almost no overhead, even if the object test resides in an assertion. Object tests are translated in the feature *process_object_test_as* of the class *SCOOP_CLIENT_FEATURE_VISITOR*. The feature takes care of the translation described in listing 2.3. Assignment attempts can only occur inside of a feature body and not inside assertions. This allows us to do a direct translation in the feature without generating overhead. The translation of assignment attempts is handled by the feature *process_reverse_as* of the class *SCOOP_CLIENT_FEATURE_VISITOR* applying the rules described in listing 2.4.

Chapter 3

Feature Redeclaration

3.1 Problem

The current[1] SCOOP implementation does not correctly process code within feature redeclarations involving separate arguments or separate return types. The following setup in code snippet 3.1 shows valid SCOOP code explaining the problem.

This code snippet shown in figure 3.1 is processed by the compiler and transformed into a client class similar to the code shown in code snippet 3.2. The generated code is not valid Eiffel code since it does not follow the Eiffel redeclaration rules. For this to be valid Eiffel code *SCOOP_SEPARATE__Y* would have to be a subtype of X, and Y would have to be a subtype of *SCOOP_SEPARATE__X*.

In general there are two cases where this problem arises:

1. Features with separate return types are redeclared into features with non-separate return types.
2. Features with a non-separate argument are redeclared into features with a separate argument.

3.2 Solution

The problem was solved by redefining how features with a redeclared ancestor version are processed. The solution to the first case is to substitute the redeclared non-separate return type in the child version of the feature into a separate type to ensure that the generated feature signature conforms to valid Eiffel code. This change is only applied on the signature of the child feature, internal representations of the return type inside the feature stay non-separate

[1] Eiffel Verification Environment with SCOOP integration

Listing 3.1: *Eiffel*: Feature redeclaration

```
1
2  class
3      A
4  feature
5      f (a: X): separate X is
6
7        do
8          ...
9        end
10 end
11
12 class
13     B
14 inherit
15     A
16     redefine
17         f
18     end
19 feature
20     f (a: separate Y): Y is
21       do
22         ...
23       end
24
25 end
26
27 class
28     X
29 end
30
31 class
32     Y
33 inherit
34     X
35 end
```

Listing 3.2: *Eiffel*: Feature redeclaration

```
1
2  class
3      A
4  feature
5      f (a: X): SCOOP_SEPARATE__X is
6
7        do
8          ...
9        end
10 end
11
12 class
13     B
14 inherit
15     A
16     redefine
17         f
18     end
19 feature
20     f (a: SCOOP_SEPARATE__Y): Y is
21       do
22         ...
23       end
24 end
```

and no adaption is needed. In order to guarantee that a local object is inserted of the original type which replaces the result internally and at the end of the feature its proxy type is assigned to the actual *result*. In addition to that a void check has to be inserted in order to deal with void results because we cannot query the proxy type of a void object. Changing the signature requires adaption on the queries which query the child feature because they expect a non-separate result. Those adaptions of the queries are done on the caller side and are achieved by converting the query result returned from the feature to its client type using the implemented query described in chapter 2. The proxy class is using local agent objects in order to call the features from the client side which is described in detail in Patrick Huber's master thesis[2]. Those local agent objects are created with the signature of the SCOOP feature and since we changed that signature an adaption of the signature of the local agent object is also needed. These adaptions are shown in code snippet 3.3 where we have processed code from a similar version as shown in 3.1 where the return type was redefined into a non-separate type and the formal argument was redefined into a separate type.

The solution of the second case is to substitute the redeclared separate argument in the child feature into a non-separate type to ensure that the generated feature signature conforms to valid Eiffel code. Different from the first case this change affects the internal representation of the argument. In order to work with the new signature we need to adapt every access to the substituted argument expecting a separate type as well as all feature calls made to the child feature. Every access to the substituted argument in the feature body is converted to their proxy type, using the implemented query described in chapter 2, in order to conform to the signature change. Feature calls on the child feature are adapted in the way that the arguments which where substituted in the signature are passed on as their client type. These adaptions are shown in code snippet 3.3 where we have processed code from a similar version as shown in 3.1 where the return type was redefined into a non-separate type and the formal argument was redefined into a separate type.

Both of the cases also apply to generic class types, where a check is needed, if it was redefined from an ancestor version, on every generic parameter.

3.3 Implementation

3.3.1 Return type substitution

The class *SCOOP_CLIENT_FEATURE_VISITOR* implements the client side return type substitution. The substitution is triggered in the feature *process_body_as* where the flag *result_substitution* is set. If the flag is set, the feature *process_routine_as* is responsible for creating the local object called *nonseparate_result* which signature is of the substituted type in order to return the result as the type specified in the changed signature. The feature *add_result_type_substitution* converts the result of the called feature to its client object on the caller side by adding *.implementation_* to it and is called whenever there is an access to the child feature.

Listing 3.3: *Eiffel*: Feature redeclaration

```
1
2  class
3      A
4  feature
5      f (a: X): SCOOP_SEPARATE__X is
6
7        do
8          ...
9        end
10 end
11
12 class
13     B
14 inherit
15     A
16     redefine
17         f
18     end
19 feature
20     f (a: Y): SCOOP_SEPARATE_Y is
21       -- Access on the internal argument 'a'
22       local
23         nonseparate_result: Y
24         ys: SCOOP_SEPARATE__Y
25       do
26         ...
27         ys := a.proxy_
28
29         if nonseparate_result /= Void then
30           Result := nonseparate_result.proxy_
31         else
32           Result := create {SCOOP_SEPARATE__Y}.
                 set_processor_ (Current.processor_)
33           Result.set_implementation_(Void)
34         end
35
36       end
37
38     g is
39       -- Access on the return type of 'f'
40       local
41         y: Y
42         ys: SCOOP_SEPARATE__Y
43       do
44         ...
45         y := f(ys.implementation_).implementation_
46       end
47 end
```

If the return type of the processed feature is of a generic type, the feature *process_body_as* takes the generic parameters which need to be substituted by calling the feature *generic_parameters_to_replace* in *SCOOP_PROXY_ASSIGN_FINDER* and passes them on to the generic visitor (*SCOOP_GENERICS_VISITOR*). The visitor processes the generic type and, if needed, substitutes the generic parameters by comparing the currently processed parameter with the ones from the list which was passed on by *process_body_as* based on the *generic_index*. The generic index describes the position of the generic parameter relative to the generic signature. The index is composed of two parts, a stage and a position. The stage defines the depth of nesting in which the generic parameter resides and the position describes the position of the generic parameter in that depth. An actual generic type of the form Y[A,X[B,C]] would result in the following generic indexes for the generic parameters:

Generic paramter	Stage	Position	Generic index
Y	1	1	[1,1]
A	2	1	[2,1]
X	2	2	[2,2]
B	3	1	[3,1]
C	3	2	[3,2]

SCOOP_SEPARATE_PROXY_PRINTER handles the proxy side substitution where we have to adapt the type of the local agent, used to call the client feature which is described in detail in Patrick Huber's master thesis[2]. This is done in the feature *process_auxiliary_local_variables* which checks if there was a signature substitution on the client side and changes the return type of the local agent object if needed. This mechanism is also applied to the features *process_attribute* and *process_function_content* in the class *SCOOP_PROXY_FEATURE_VISITOR*. Without result type substitution, the result returned from the agent has to be converted to its proxy type because of the fact that the return type of the feature is separate on the proxy side and we get a non-separate result back from the agent. If we apply the result substitution this reconversion has to be skipped because we already get a separate result back from the client side which is denoted by the flag *add_result_substitution*.

The class *SCOOP_PROXY_ASSIGN_FINDER* handles the discovery of ancestral versions of the feature in the feature *have_to_replace_return_type*. It triggers a traversal of the class hierarchy starting at the class and feature given as arguments trying to find an ancestral version of the feature that possesses a separate return type. This is needed in order to check if there was an ancestor version of a feature with a signature that does not conform to the signature of the child feature which would lead to an insertion of the solution.

3.3.2 Actual argument substitution

The class *SCOOP_CLIENT_FEATURE_LR_VISITOR* implements the actual argument substitution since the substitution is only applied if the child feature redeclares a parameter to a separate type. The substitution is handled by the feature *single_process_identifier_list* which processes each argument individually

and compares them to ancestral versions by calling *need_internal_argument_substitution* of the class *SCOOP_CLIENT_FEATURE_VISITOR*. If there is an argument which needs to be substituted in the current feature, it is added to the *internal_arguments_to_substitute* list which resides in the *feature_object*, in order to be able to remember which arguments were substituted. The access conversion inside the child feature is done by the feature *process_id_as*, where every processed identifier is compared with the elements in the internal arguments to substitute list and, if matched, converted to its proxy version. The conversion of the arguments used to call the child feature is done in the feature *process_parameter_list_as* of the class *SCOOP_CLIENT_FEATURE_VISITOR* where every argument is compared to its ancestral versions and, if needed, converted to their client.

If the argument type of the processing feature is of a generic type, the feature *single_process_identifier_list* receives the generic parameters which needs to be substituted by calling the feature *generic_parameters_to_replace* in *SCOOP_PROXY_ASSIGN_FINDER* and passes them on to the generics visitor (*SCOOP_GENERICS_VISITOR*). The visitor processes the generic type and substitutes the generic parameters by comparing the currently processed parameter with the ones from the list which was passed on by *process_body_as* based on the *generic_index*. The generic index describes the position of the generic parameter relative to the entire generic signature as it was described in 3.3.1.

The implementation on the proxy side is done in the class *SCOOP_SEPARATE_PROXY_PRINTER* where we have to adapt all feature calls made to the client version of the feature if some internal argument was substituted. The feature *process_formal_argument_list_with_auxiliary_variables* makes sure that every argument passed to the client side is compared to the *internal_arguments_to_substitute* list in the *feature_object* and, if matched, converted to its client version.

The class *SCOOP_PROXY_ASSIGN_FINDER* handles the discovery of ancestral versions of the feature in the feature *have_to_replace_internal_arguments*. It triggers a traversal of the class hierarchy starting at the class and feature given as arguments trying to find an ancestral version of the feature with the internal argument as a non-separate type. This is needed in order to check if there was an ancestor version of a feature with a signature that does not conform to the signature of the child feature which do lead to an insertion of the solution. .

Chapter 4

Create expressions

4.1 Problem

The current[1] SCOOP implementation does not correctly process code within creation instructions and creation expressions. In both cases the creation of the object is not correctly translated and transformed into processed Eiffel code. The assignment of the correct processor to the created object is not invoked in the current implementation. In the case of creation instructions an implementation which was done by Patrick Huber[2], was already present. That implementation had to be reworked to match the work on creation expressions and to reflect processors handling the created objects of non-separate nature. Dealing with creation expressions we need to take into account that those creations can appear inside assertions which call for a more sophisticated solution in order to initialize the created object correctly because we are not able to insert instructions inside contracts.

4.2 Solution

In principle, the creation and initialization of the new object is the same in creation instructions and creation expressions. The translation and transformation of the creation instructions and creation expressions into Eiffel code needs to be handled differently though.

Creation instructions are standalone, meaning that they cannot be in-lined into other expressions with the exceptions of in-line agents. This allows us to insert other instructions directly in front of the object creation into Eiffel code without having to create overhead in form of a wrapper. In order to translate creation instructions into Eiffel code we need to deal with four kinds of object creations and translate them accordingly. In SCOOP created objects can be of the following nature: non-separate, separate with no explicit processor specifications, separate with a qualified explicit processor specification or separate

[1] Eiffel Verification Environment with SCOOP integration

with an unqualified explicit processor specification. Code snippet 4.2 shows how those four cases look in SCOOP code, and snippet 4.3 shows how they are translated into Eiffel code.

Creation expression can appear in other expressions, in feature calls, even in contracts. This requires a more sophisticated translation of the creation expression because we cannot just insert code the same way as in the case of creation instructions. A local placeholder object is created with the same type as the original object which is initialized and created just before the expression that encloses the original creation expression, if possible. Instead of the creation expression itself the placeholder object or the wrapper query is then passed on in the expression which enclosed the original creation expression. Special care is taken when processing a creation expression inside an assertion because we are not able to induce code right before an assertion. A wrapper which handles the creation and initialization of the object is created in that case as shown in code snippet 4.1. The assertion expression then calls the created wrapper instead of the creation expression. The core translation of the object creation stays the same in any of the cases and is handled as shown in code snippet 4.3.

Creation instructions and creation expressions have the same motivation in their translations which is different depending if the created object is separate or non-separate as well as if there is a specific processor specification defined. The following cases exhibit the diversity of the events which can occur during the creation of new objects in SCOOP. A wild card (*) was inserted for the attachment symbol, it may vary, but does not have any influence on the translation:

- *Object (*, non sep, ANY)*

  A non-separate object is created. We create the object by using the SCOOP infused creation procedure *set_processor_*, once the object is created we call the creation procedure on the object, if there is any.

- *Object (*, <e.handler>, ANY)*

  A separate object with a qualified explicit processor specification is created. After making sure that 'e' is not of *void* type and possesses a processor, we create the object by using the SCOOP infused creation procedure *set_processor_*. The creation procedure creates the object and assigns the processor given as argument, in this case the processor of 'e', to the created object. After creating the object the feature *separate_execute_routine* (described below) executes either the default creation procedure or the specified creation procedure of 'ANY'.

- *Object (*, <p>, ANY)*

  A separate object with an unqualified explicit processor specification is created. If 'p' is of *void* type we assign a new processor given by the SCOOP scheduler to it and we create the object by using the SCOOP infused creation procedure *set_processor_*. The creation procedure creates the object and assigns the processor given as argument, in this case

Listing 4.1: *Eiffel*: Creation expression wrapper

```
1  class
2    APPLICATION
3  ...
4  make_application_sp_create_creation_wrapper_nr1:
       SCOOP_SEPARATE__X is
5      -- Wrapper for a separate creation expression
6      local
7        creation_object:  SCOOP_SEPARATE__X
8      do
9        create creation_object.set_processor_ (scoop_scheduler
             .new_processor_); separate_execute_routine ([
             creation_object.processor_], agent creation_object.
             make_scoop_separate_x(Current, True), Void, Void,
             Void)
10       Result := creation_object
11     end
12
13   make_application_sp_create_creation_wrapper_nr2(x_arg:
         SCOOP_SEPARATE_TYPE): SCOOP_SEPARATE__X is
14     -- Wrapper for separate creation expression with a
           qualified explicit processor specification
15     local
16       creation_object: SCOOP_SEPARATE__X
17     do
18       check x_arg /= Void and then x_arg.processor_ /= Void
             end; create creation_object.set_processor_ (x_arg.
             processor_); separate_execute_routine ([
             creation_object.processor_], agent creation_object.
             make_scoop_separate_x(Current, True), Void, Void,
             Void)
19       Result := creation_object
20     end
21
22     make_application_sp_create_creation_wrapper_nr3:
           SCOOP_SEPARATE__X is
23     -- Wrapper for separate creation expression with an
           unqualified explicit processor specification
24     local
25       creation_object: SCOOP_SEPARATE__X
26     do
27       if p = Void then p := scoop_scheduler.new_processor_
             end; create creation_object.set_processor_ (p);
             separate_execute_routine ([creation_object.
             processor_], agent creation_object.
             make_scoop_separate_x(Current, True), Void, Void,
             Void)
28       Result := creation_object
29     end
30   ...
```

the processor of 'e', to the created object. After creating the object, the feature *separate_execute_routine* (described below) executes either the default creation procedure or the specified creation procedure of 'ANY'.

- *Object (*, sep, ANY)*

  A separate object with no explicit processor specification is created. We assign a new processor given by the SCOOP scheduler to the object and then call the feature *separate_execute_routine* (described below) which executes either the default creation procedure or the specified creation procedure of 'ANY'.

The feature *separate_execute_routine* pauses until two conditions are met atomically. The locks of the processors given in the argument *a_requested_processors* must be acquired on behalf of the current processor and the wait condition given in the argument *a_wait_condition* is satisfied. Then the current processor executes the enclosing routine given in *a_routine*. Once this is completed, the current processor executes the non-separate part of the postcondition given in *a_non_separate_postcondition*. Furthermore, the current processor executes the separate part of the postcondition given in *a_separate_postcondition*. If the current processor detects that it is not involved in the evaluation of the separate part of the postcondition, then the separate part of the postcondition will be executed asynchronously by other processors.

In the previous implementation the creation of a proxy object did not create a client object right away. Instead the client object got created asynchronously by means of a creation routine call on the client object. This could lead to void calls on the implementation of the proxy while the asynchronous creation routine call was pending. We resolved this issue by creating an empty object with the default constructor and then we call the actual constructor on the created object asynchronously.

4.3 Implementation

The class *SCOOP_CLIENT_CONTEXT_AST_PRINTER* implements both, creation instructions as well as creation expression.

The feature *process_create_creation_as* implements the core aspects of creation instructions. Creation instructions are processed depending on the target type unless they contain a specific type definition, then they are processed according to their specified type.

Creation expressions are implemented in the feature *process_create_creation_expr_as* and are processed according to their type. The list *create_creations* in the derived class information keeps track of the creation expressions in the entire class in order to track which creation expressions have already been processed so we do not create multiple instances of a wrapper while processing the creation expression multiple times. To be able to produce uniquely identifiable wrappers we keep the position of the current creation expression relative to all the creation expressions appearing in the class and encode this position in the wrapper name.

Listing 4.2: *Eiffel*: Eiffel Creation

```
1
2  class
3      A
4  feature
5      make is
6        local
7          x : X
8          sx: separate X
9          sxh: <sx.handler> X
10         sxp: <p> X
11
12       do
13         create x
14         create sx
15         create sxh
16         create sxp
17
18       end
19     p: PROCESSOR
20 end
```

Following indexes have to be kept and updated in the feature object of the currently processed feature and the derived class information:

- *feature_object.locals_index* defining the position used to insert local objects
- *feature_object.last_instr_call_index* defining the position used to insert creations before the enclosing expression
- *derived_class_information.wrapper_insertion_index* defining the position used to insert uniquely identifiable wrappers

Listing 4.3: *Eiffel*: SCOOP Creation

```
1
2  class
3      A
4  feature
5      make is
6        local
7          x : X
8          sx: separate X
9          sxh: <sx.handler> X
10         sxp: <p> X
11
12       do
13         create x.set_processor_(Current.processor_);
14         create sx.set_processor_ (scoop_scheduler.
               new_processor_); separate_execute_routine ([ sx.
               processor_], agent  sx.
               default_create_scoop_separate_x(Current), Void,
               Void, Void)
15         check sx /= Void and then sx.processor_ /= Void end
               create sxh.set_processor_ (sx.processor_);
               separate_execute_routine ([ sxh.processor_],
               agent  sxh.default_create_scoop_separate_x(
               Current), Void, Void, Void)
16         if p = Void then p:= scoop_scheduler.new_processor_
               end; create sxp.set_processor_ (p);
               separate_execute_routine ([ sxp.processor_],
               agent  sxp.default_create_scoop_separate_inv (
               Current), Void, Void, Void)
17
18       end
19     p: PROCESSOR
20 end
```

Chapter 5

Agent creation

5.1 Problem

The current[1] SCOOP implementation does not support agent or in-line agent creation correctly. The current implementation does not assign processors to created agent and in-line agent objects. The agent and in-line agent object have to be created with the correct processor assigned to them. The correct processor for agent objects is deducted from the target, if existent, or from the *current* object for in-line agents.

5.2 Solution

Dealing with agents the implementation has to involve the target of the agent, if there is one. If the target of an agent object is separate the implementation has to compute the target's processor and assign the agent object to it. In-line agents on the other hand never possess a target definition which requires them to be assigned to the processor of the *current* object. To achieve the correct binding of the processor to the agent or in-line agent object we create a local object, assign the correct processor to it and pass it on instead of the original agent or in-line agent object. As described before in-line agents are always created on the processor handling the *current* object. For agent objects a check is needed if there is an existing target and, if so, the local agent object has to be assigned to the target's processor. Creating a local object is not possible if an agent resides in a contract because we cannot insert instructions before an assertion, in that case the solution creates a wrapper the same way it is handled by the creation expressions shown in code snippet 4.1.

[1] Eiffel Verification Environment with SCOOP integration

5.2.1 Implementation

The implementation of in-line agents is located in the feature *process_inline_agent_creation_as* of the class *SCOOP_CLIENT_FEATURE_VISITOR*. The feature creates a local in-line agent object which signature is generated from the internal arguments and the return type of the in-line agent in order to assign the processor of the *current* object to it. If the in-line agent resides in an assertion a wrapper is created to enclose the creation code because we are not able to insert instructions inside of contracts. The created local agent object or the wrapper is then passed on instead of the original in-line agent.

The feature *process_agent_routine_creation_as* of the class *SCOOP_CLIENT_FEATURE_VISITOR* implements the processing of agents. A local agent object is created which signature is generated from the internal arguments and the return type of the agent definition. In case there is an existing target in the agent definition, the target's processor is assigned to the local object. If there is no target defined, the *current* object's processor is assigned to the local agent object. In the case of nested agents inside an in-line agent we have to define the local object in the in-line agent's *local* clause, because we cannot use local variables defined in the enclosing feature inside an in-line agent. This requires that we keep track of the latest local definition in case of nested in-line agents as well as for agents residing inside of the in-line agent. If the agent resides inside an assertion a wrapper is created to enclose the creation code because we are not able to insert instructions inside of contracts.

5.3 To do

In order to make this implementation work some problems have to be addressed. The main problem arises due to the nature of the signature of an agent object. We are forced to SCOOP process the class *PROCEDURE*, *ROUTINE* and *FUNCTION* in order to deal with agents on the proxy object. To be able to SCOOP process those classes about eighty Eiffel base classes would have to be SCOOP compiled, involving a great amount of work to get them to be processed by SCOOP, exceeding the limits of this master thesis. Another problem arises when processing agents inside assertions. To handle agent objects inside assertions a similar mechanism as the one described in chapter 4, handling create creation expressions inside assertions, is needed in order to create wrapper feature for the local object creation.

Chapter 6

Conclusions

6.1 Conclusions

We were able to improve the current[1] SCOOP implementation considerably by implementing some of the missing advanced SCOOP aspects described in this paper. By adding support for feature redeclaration shown in chapter 3, create creation of chapter 4 and up- and downcast from chapter 2, we are now able to support more sophisticated projects programmed in SCOOP than before involving those aspects. A step closer to achieve a complete SCOOP implementation in EVE was made by successfully implementing and allowing those advance aspects of SCOOP in the current implementation.

6.2 Future Work

During the development of code, processing agent and in-line agent expressions, we attempted to SCOOP process the needed Eiffel base classes in order to be able to handle agent and in-line agent expressions. Due to the complex nature of the Eiffel base classes a lot of distinctive cases of problems surfaced while processing them, which are not taken care of in the current SCOOP implementation, i.e. handling of alias expressions. The completion of the agent and in-line agent creation implementation needs to be pursued which requires to be able to SCOOP process the Eiffel base classes. In order to do that there is the need to complete the current SCOOP implementation in regards of advanced SCOOP aspects which will enable the processing of Eiffel base classes under SCOOP. In order to fully support agent and in-line agent creations we need to adapt the algorithm to support agent creations in assertions adapting and enhancing the current implementation with the mechanisms as described in chapter 4.

While implementing feature redeclaration regarding return types respectively internal arguments an incorrectness of the type checker was found residing

[1] Eiffel Verification Environment with SCOOP integration

in the feature *scoop_conform_to* of the class *CL_TYPE_A*. Part of the responsibility of the boolean expression *lte_proc_tag* is to make sure that redefined return types as well as redefined internal arguments conform to the type of the ancestral feature. The boolean expression as it stands today is only correctly applied for return types but not to internal arguments. In order to be able to compile redefinitions of internal arguments this boolean expression was commented. Future work should include altering the boolean expression to support redeclaration of internal arguments or to disable the conformance check for internal arguments, to be able to support redeclaration of internal arguments and to re-enable this conformance rule.

Bibliography

[1] Piotr Nienaltowski, *Practical framework for contract-based concurrent object-oriented programming*, http://se.inf.ethz.ch/people/nienaltowski/papers/thesis.pdf

[2] Patrick Huber, *Integrating SCOOP into EVE*, http://se.inf.ethz.ch/projects/patrick_huber

[3] Eiffel Verification Environment (EVE), http://eve.origo.ethz.ch

[4] Simple concurrent object oriented programming (SCOOP), http://scoop.origo.ethz.ch