# Chapter 4

# Combined Operators

The operators that were introduced in chapter 2 are too rudimentary to model a real software system. Therefore we need a means of combining these primitive operators into more complex ones. Experiments show that some combinations appear so often that it is useful to explicitly introduce an operator for them. In this chapter we first provide some general definitions on how to combine operators and then define a number of combined operators: extension, refinement, connected extension, extending refinement and factorisation. These are not the only useful combinations, others can be defined at will.

## 4.1 Composition of Modifiers

### 4.1.1 Applicability

To enable a more legible formulation of the rules we first introduce one extra definition, with two new expressions.

**Definition 4.1 (Single Modifier Applicability & Result Of)** A reuse modifier $M$ **is applicable to** a reuse contract $R$ and $R'$ **is the result of applying** $M$ to $R$ in any of the following cases:

1. $M$ is a participant extension modifier, $R$ is participant extendible with $M$ and $R'$ is the participant extension of $R$ with $M$ ;

2. $M$ is a context extension modifier, the context of $R$ is extendible with $M$ and $R'$ is the context extension of $R$ with $M$ ;

3. $M$ is a participant cancellation modifier, $R$ is participant cancellable with $M$ and $R'$ is the participant cancellation of $R$ with $M$ ;

4. $M$ is a context cancellation modifier, the context of $R$ is cancellable with $M$ and $R'$ is the context cancellation of $R$ with $M$ ;

5. $M$ is a participant refinement modifier, $R$ is participant refinable with $M$ and $R'$ is the participant refinement of $R$ with $M$ ;

6. $M$ is a context refinement modifier, the context of $R$ is refinable with $M$ and $R'$ is the context refinement of $R$ with $M$ .

7. $M$ is a participant coarsening modifier, $R$ is participant coarsenable with $M$ and $R'$ is the participant coarsening of $R$ with $M$ ;

8. $M$ is a context coarsening modifier, the context of $R$ is coarsenable with $M$ and $R'$ is the context coarsening of $R$ with $M$ .

### 4.1.2 Definition and Properties

Based on the above definition we can now define in general how a sequence of modifiers can be composed. A reuse modifier of a combined operator again has a modifier tag. The modifier description has the form of a sequence of other reuse modifiers. We use the terminology "sequence" here, because the order in which the modifiers are combined is usually important.

**Definition 4.2 (Combined Reuse Modifier)** A **combined reuse modifier** consists of a modifier tag and a sequence of reuse modifiers.

Note that this definition implies that reuse modifiers can be nested.
We can now define applicability of combined modifiers.

**Definition 4.3 (Combined Modifier Applicability & Result Of)** A combined modifier $M = (tag, (M_1, \ldots, M_n))$ **is applicable to** a reuse contract $R_0$ if
$\forall i : M_i$ is applicable to $R_{i-1}$, with $R_i$ as result
$R_n$ is then called **the result of applying** $M$ to $R_0$.

For sequences of length one, this definition reduces to the definition of applicability of section 4.1.1.

No well-formedness checks are needed in definition 4.3, because well-formedness is checked in the applicability of each of the composing modifiers. The following property can be proven in this regard.

**Property 4.1** *If $R$ is a well-formed reuse contract, $M$ is a combined reuse modifier and $R_r$ is the result of applying $M$ to $R$; then $R_r$ is well-formed*

**Proof**

This property follows immediately from the definition, as $R_r$ is the result of applying a sequence of modifiers that are all applicable to the intermediary reuse contracts. As applicability of all primitive operators preserves well-formedness of the resulting reuse contract, all these intermediary reuse contracts are well-formed, with as a special case $R_r$.

### 4.1.3 Discussion

It was demonstrated in the previous chapter that in most cases conflicts that occur due to one modifier in a chain also represent a conflict in the combined operator. A number of exceptions were formulated, where one modifier annihilates the effects (and thus the possible conflicts) of another modifier. Here, we will also see an example of a situation where the effects caused by two or more composing modifiers are not annihilated at the level of the combined operator, but the conflicts they cause can be neglected because of the meaning of the combined operator. Therefore we add a section on the 'Impact on the Conflicts' to every definition of a combined operator.

The opposite situation, where at the level of the combined operator extra conflicts arise that do not correspond to lower level conflicts, does not occur. The reason for this is that the combined operators do not impose any extra constraints on the original reuse contract they are applied to. The definitions of the combined operators only impose constraints on how the combining operators relate to each other. When these constraints are broken, the combination simply no longer forms the intended combined operator.

Because the applicability of the composing modifiers is expressed straightforwardly by means of the applicability definition for sequences of modifiers, no extra applicability constraints are needed for the combined operators. Therefore, for each combined operator we only state a definition of the combined modifier and the definition of the operator. Moreover, this latter definition always has the same form and is thus very straightforward. Furthermore, as we have proven the preservation of well-formedness above for the general case, we do not need to prove it for each operator separately.

## 4.2 Extension and Refinement

**Motivation**

The simplest combined operator is a combination of a number of operators of the same kind. We thus define extension as a combination of participant and context extensions and refinement as a combination of participant and context refinements.

**Definition and Properties**

As the structure of the combined modifiers representing extension and refinement is very simple, we immediately define the operators.

**Definition 4.4 (Extension)** $R_e$ is an **extension** of R with $M$ = ("extension", $(M_1, \ldots, M_n)$) if

1. $M_1$, …,$M_n$ are participant extension and context extension modifiers;

2. $M$ is applicable to $R$ ;

3. $R_e$ is the result of applying $M$ to $R$ .

Refinement can be defined in a completely similar fashion.

**Definition 4.5 (Refinement)** $R_r$ is a **refinement** of R with $M =$ (“refinement”, $(M_1, \ldots, M_n)$) if

1. $M_1, \ldots, M_n$ are participant refinement and context refinement modifiers;

2. $M$ is applicable to $R$ ;

3. $R_r$ is the result of applying $M$ to $R$

Completely similar definitions can also be given for coarsening and cancellation.

Note that because an extension also includes context extensions, some of the participant extensions above can be performed on participants that are newly introduced by one of these context extensions. Similarly, participant refinements can be performed on acquaintance relationships added through context refinements. Note, however, that for this to be possible the context modifier has to precede the participant modifier in the sequence.

**Impact on Conflicts**

No conflicts are annihilated by these combinations, so the conflicts that can occur after the application of extension is the union of those that can appear after participant extension and those that can appear after context extension. The same holds for refinement. Table 4.1 gives an overview of the possible conflicts. This table is made by taking the union of the relevant parts of tables 3.1, 3.2 and 3.3 from chapter 3. Note that to detect unanticipated recursion, the rules discussed in chapter 3 considering transitive closure should be used.

Note that, unless explicitly mentioned otherwise, when in the following sections we define operators combined from extensions, refinements and coarsenings, we imply the general operators, i.e., the combinations of participant *and* context extensions, refinements and coarsenings. This is possible, because combined modifiers are allowed to be nested. In the examples, we sometimes show only one level.

## 4.3 Connected Extension

**Motivation**

In the basic operators the extension is self-contained, i.e., the newly added operations only contain operations added by the same extension in their specialisation clause. Of course, one also wants to be able to add references to already existing

| | Extension | | Refinement | |
|---|---|---|---|---|
| | part. ext. | cont. ext. | part. ref. | cont. ref. |
| part. ext. | oper. name, acc. oper. capt., unant. recursion | - | - | - |
| cont. ext. | - | part. name | - | - |
| part. canc. | - | - | dang. oper. ref. | - |
| cont. canc. | dang. part. ref. | - | dang. part. ref. | dang. part. ref. |
| part. ref. | - | - | oper. invoc., unant. recursion, reg. oper. capt. | - |
| cont. ref. | - | - | - | acq. rel. |
| part. coars. | - | - | oper. invoc., unant. recursion, reg. oper. capt. | - |
| cont. coars. | dang. acq. ref. | - | dang. acq. ref. | acq. rel. |

Table 4.1: Conflicts with Extension and Refinement

operations to the specialisation clauses of the newly added operations. This can be achieved by first performing an extension and then a refinement that augments the specialisation clauses of the newly added operations with operations from the original contract only. In a similar vein, one can first perform a context extension adding new participants and then a context refinement connecting those newly added participants to participants from the original contract. This is called a *connected extension.*

An example of a connected extension on the context level is given in figure 4.1. A participant `Bank` is added through an extension and an acquaintance relationship between this new participant and the already existing participant `Consortium` is added through a context refinement. Note the direction of the acquaintance relationship: from the newly added participant to the existing one. The other direction would not imply a connected extension.

## Definition and Properties

**Definition 4.6 (Connected Extension Modifier)** A **connected extension modifier** is a combined reuse modifier with modifier tag "connected extension" and a modifier description containing a sequence $(M_e, M_r)$, where:

1. $M_e$ is an extension modifier and $M_r$ a refinement modifier;
2. (a) $M_r$ only adds operation invocations to the specialisation clauses of operations that were added by $M_e$;

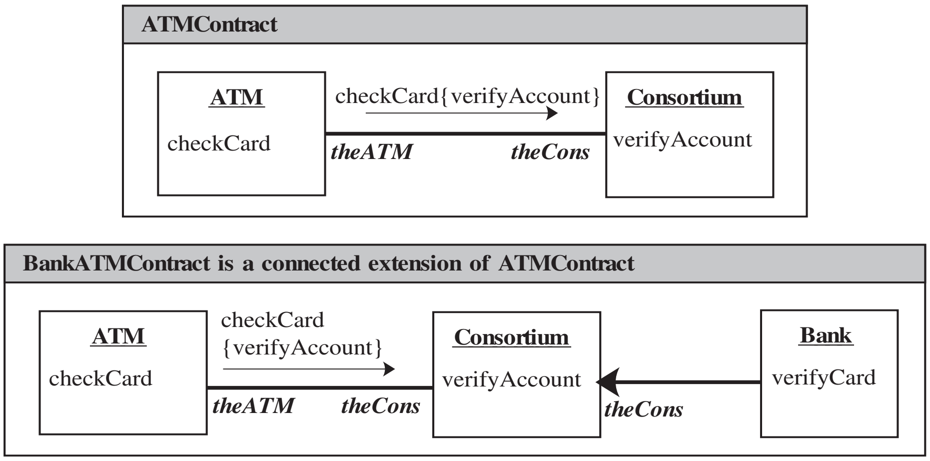


Figure 4.1: An Example Connected Extension

   (b) $M_r$ only adds acquaintance relationships to the acquaintance clauses of participants that were added by $M_e$.

3. (a) for each $a.m$ added to a specialisation clause of an operation in the interface of $p$ by $M_r$: $m$ is not in the interface of $p$ in $M_e$;

   (b) for each $a.q$ added to an acquaintance clause of a participant $p$ by $M_r$: $q$ is not in $M_e$.

Clause 2 ensures that the refinements are only applied to newly added items, while 3 ensures that the refinements only add dependencies on items that already existed.

**Definition 4.7 (Connected Extension)** $R_{ce}$ is a **connected extension** of $R$ by $M$ if

1. $M$ is a connected extension modifier;
2. $M$ is applicable to $R$ ;
3. $R_{ce}$ is the result of applying $M$ to $R$

**Illustration**

Figure 4.2 illustrates a connected extension. First, an extension is performed adding a new participant, pcpt3, and a new operation on pcpt1, $name_{12}$. Note that this is an example of an extension on both participant and context level. Then a refinement is performed, again on both levels, that refines these two newly added items.

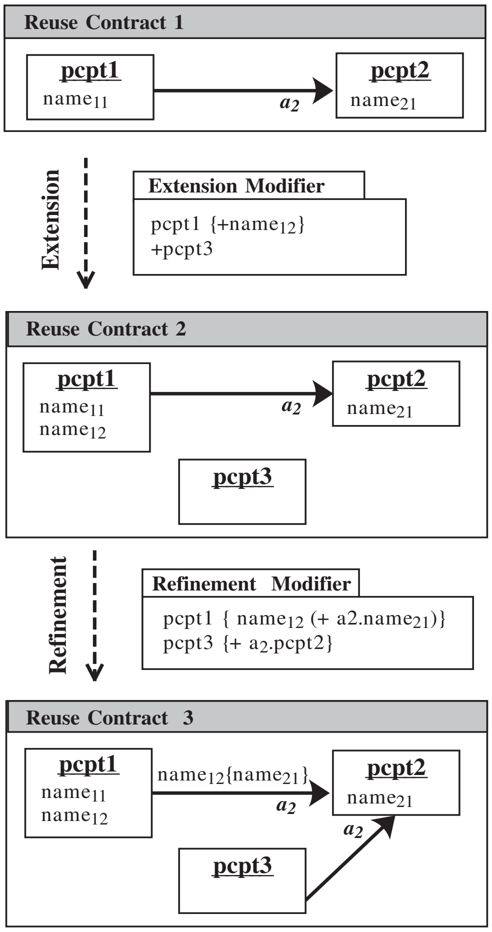


Figure 4.2: Connected Extension

The specialisation clause of $name_{12}$ is augmented with an invocation of $name_{21}$ on pcpt2 and the acquaintance clause of pcpt3 is augmented, with an acquaintance name $a_2$ referring to pcpt2.

Note that, as stated by clauses 2 and 3 of the connected extendibility definition, only items added through the extensions are refined (pcpt3 and $name_{12}$) and the refinement only adds dependencies on items already existing in the original contract (pcpt2 and $name_{21}$).

### Impact on Conflicts

The conflicts that can occur after the application of connected extension are the sum of those that appear after extension and refinement separately. Table 4.2 gives an overview of the possible conflicts. This table is again constructed by taking the union of the relevant parts of tables 3.1, 3.2 and 3.3 from chapter 3.

| | Connected Extension | | | |
|---|---|---|---|---|
| | part. ext. | part. ref. | cont. ext. | cont. ref. |
| part. ext. | oper. name, acc. oper. capt., unant. recursion | - | - | - |
| cont. ext. | - | - | part. name | - |
| part. canc. | - | dang. oper. ref. | - | - |
| cont. canc. | dang. part. ref. | dang. part. ref. | - | dang. part. ref. |
| part. ref. | - | oper. invoc., unant. recursion, reg. op. capt. | - | - |
| cont. ref. | - | - | - | acq. rel. |
| part. coars. | - | oper. invoc., unant. recursion, reg. oper. capt. | - | - |
| cont. coars. | dang. acq. ref. | dang. acq. ref. | - | acq. rel. |

Table 4.2: Conflicts with Connected Extension

## 4.4 Extending Refinement

### Motivation

Currently, a refinement can only extend specialisation and acquaintance clauses with references to operation and participant names that already exist on the base reuse contract. Sometimes one wants to simultaneously add extra operations or participants that can be referred to in the refinement. One thus wants to add an extending part to the refinement. We call this an extending refinement.

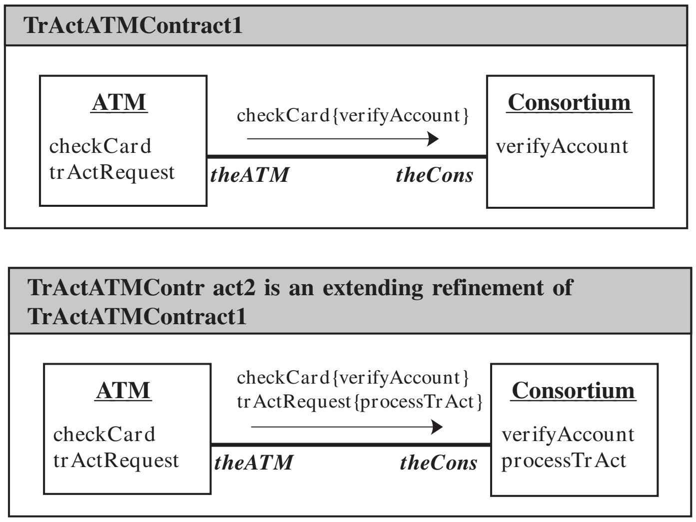


Figure 4.3: An Example Extending Refinement

Figure 4.3 shows an extending refinement on the participant level. The operation `trActRequest` on `ATM` is refined with an invocation of `processTrAct` on `Consortium`. Using the same reuse operator this operation `processTrAct` is added to `Consortium`.

### Definition and Properties

**Definition 4.8 (Extending Refinement Modifier)** An **extending refinement modifier** is a combined reuse modifier with modifier tag "extending refinement" and a modifier description containing a sequence $(M_e, M_r)$, where:

1. $M_r$ is a refinement modifier and $M_e$ is an extension modifier;
2. (a) for all operation invocations $a.m$ , added to specialisation clauses by $M_r$, with $a$ referring to $q$ : $m$ is added to $q$ by $M_e$;

   (b) for all acquaintance relationships $a.q$ , added to acquaintance clauses by $M_r$, $q$ is added to $R$ by $M_e$;
3. (a) for all operations $m$ added to $q$ by $M_e$: an operation invocation $a.m$ , where $a$ refers to $q$ is added to a specialisation clause by $M_r$;

   (b) for all participants $q$ added by $M_e$: an acquaintance relationship $a.q$ , is added to an acquaintance clause by $M_r$.

Clause 2 ensures that the refinements only add dependencies on items that already existed in the original contract, while clause 3 ensures that all items that are added are referred to by the refinement.

**Definition 4.9 (Extending Refinement)** $R_{er}$ is an **extending refinement** of $R$ by $M$ if

1. $M$ is an extending refinement modifier;
2. $M$ is applicable to $R$ ;
3. $R_{er}$ is the result of applying $M$ to $R$ .

**Illustration**

Figure 4.4 illustrates an extending refinement. First, a new participant and operation are added through an extension. Second, a participant and an operation that already existed in the original contract are refined. As demanded by clause 2 these refinements only add dependencies on the items added by the extension. Note that, as required by clause 3 of the extending refinement definition, *all* items added through the extension are referred to by the refinement. Otherwise, this would just be a basic extension.

Also note that extension has to be applied first, for extending refinement as well as for connected extension, otherwise the intermediary contract would not be well-formed. However, in the first case the refinement refines operations that were added through the extension, while in the second case the refinement refines operations that already existed before the extension.

**Impact on Conflicts**

The conflicts that can occur after the application of extending refinement are the union of those of its parts. They are thus the same set of conflicts as for connected extension, only they appear within different parts. They were already depicted in table 4.2.

## 4.5 Factorisation

**Motivation**

The third combined operator we discuss is called factorisation. Note that we do not mean factoring out common behaviour here. Intuitively, with factorisation we mean factoring out some part of the behaviour of one operation into an intermediary operation. This is achieved by replacing an operation name in a specialisation clause by another operation name, the specialisation clause of which in turn contains the originally removed operation. The net result is that the same operations that were

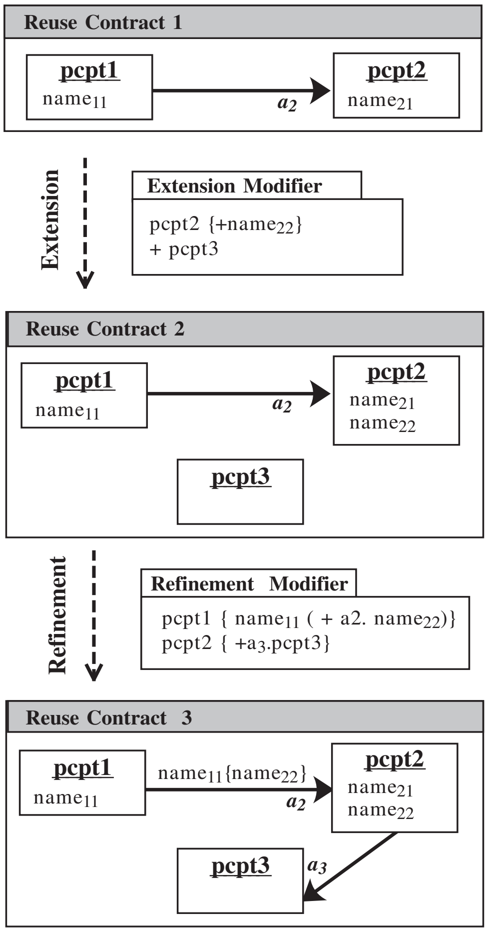


Figure 4.4: Extending Refinement

originally invoked are still invoked, but now through an extra indirection. This indirection can be added to the same participant, as well as to another participant. We only define factorisation on participant level here. A similar operator can be constructed at context level, but that would lead us too far for what we want to do here. Note that contrary to the other operators, factorisation does not really alter the behaviour of a system. It just makes the design more modular, and often better adaptable.

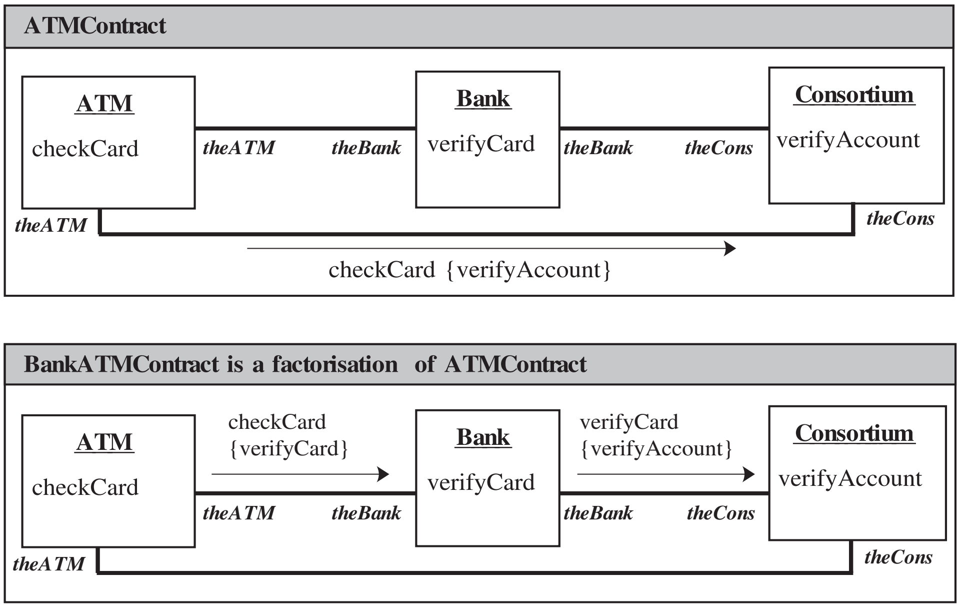


Figure 4.5: An Example Factorisation

Figure 4.5 shows an example of a factorisation. It is a factorisation because the invocation of `verifyAccount` on `Consortium` by `checkCard` on `ATM` now transits through `verifyCard` on `Bank`. Note that factorisation does not necessarily have to pass over another participant. It can also go via intra-participant invocations.

## Definition and Properties

**Definition 4.10 (Factorisation Modifier)** A **factorisation modifier** is a combined reuse modifier with modifier tag "factorisation" and a modifier description containing the sequence $(M_{ce}, M_c, M_r)$, where:

1. $M_{ce}$ is a participant connected extension modifier, $M_c$ a participant coarsening modifier and $M_r$ a participant refining modifier;

2. the set of operations that are coarsened by $M_c$ is equal to the set of operations that are refined by $M_r$;

3. for all operation invocations $a.m$ , removed from specialisation clauses by $M_c$, with $a$ referring to $q$ : operation invocations $b.m$ are added by $M_{ce}$, with $b$ referring $q$ [1];

4. for all operation invocations $b.m$ , with $b$ referring $q$ , added by $M_{ce}$: an operation invocation $a.m$ , where $a$ refers to $q$ is removed from a specialisation clause by $M_c$;

5. for each operation invocation $a.n$ added to the specialisation clause of an operation $m$ by $M_r$: $n$ is an operation in $M_{ce}$, which holds all operation names in its specialisation clause that were removed from the specialisation clause of $m$ by $M_c$;

Factorisation can then be defined as follows.

**Definition 4.11 (Factorisation)** $R_f$ is a **factorisation** of $R$ by $M$ if

1. $M$ is a factorisation modifier;

2. $M$ is applicable to $R$ ;

3. $R_f$ is the result of applying $M$ to $R$ .

**Illustration**

Figure 4.6 illustrates a factorisation. Comparing reuse contracts 1 and 4 in the figure allows to see the result of a factorisation. The communication between pcpt1 and pcpt2, i.e., the invocation of $name_{21}$ of $name_{11}$, is preserved, but now it goes through an intermediary participant pcpt3 and an intermediary operation $name_{31}$.

This result is achieved in three steps. First, a connected extension adds the intermediary operation and the operation invocation that is copied from participant pcpt1. Second, through a coarsening, the behaviour that pcpt3 has copied from pcpt1 (i.e., the invocation of $name_{21}$ on pcpt2, is removed from pcpt1). Third and last, through a refinement, a new operation invocation is added to pcpt1 to invoke the operation that has taken over its behaviour. It thus re-establishes the original behaviour through an indirection.

Note that the same result could be achieved by first applying the coarsening and then the refining extension. Even other combinations are imaginable with, for example, a pure participant extension and a refinement that adds the behaviour it adds now, as well as the refining part of the connected extension. The only requirement is that for applicability reasons the refinement should be performed after the (connected) extension. We chose the given combination because we find it more intuitive. The combination used to define factorisation is not so important, because

[1] Note that $a$ and $b$ can be identical as well as different.

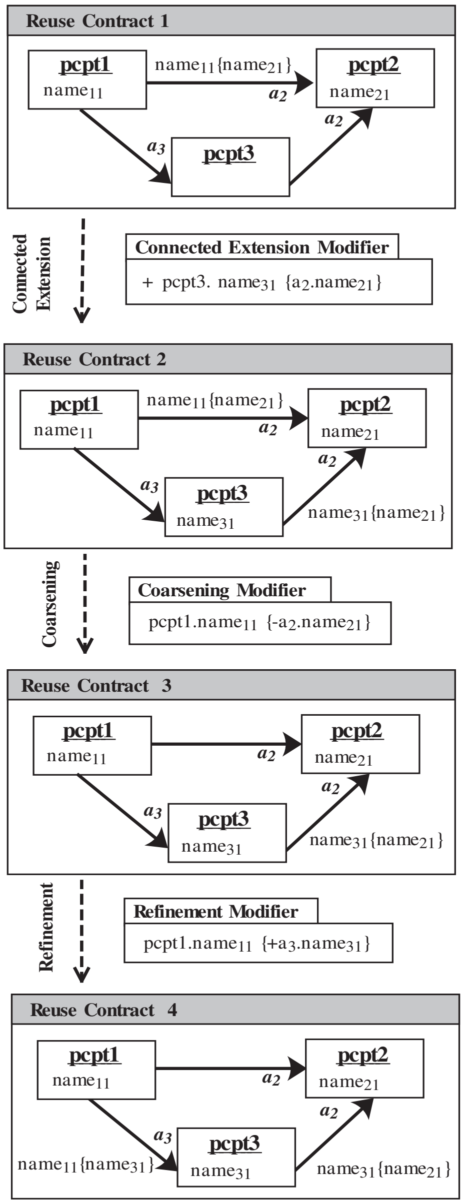


Figure 4.6: Factorisation

it is always possible to transform one combination into another one and there is no difference in the detected conflicts. What matters is that the final result is the same. Note that the definition of this operator uses the fact that combined operators can be nested. It could of course have been done with basic operators only[2], but the fact that combined operators can be nested allow for a more intuitive definition.

Let us now take a closer look at the definition. The first clause is straightforward. Clauses 2 to 5 put restrictions on the way the different sub-operators fit together, with as a goal to perform a factorisation and nothing else. One does, for example, not want the refinement to refine any other operations, than those that are strictly necessary for the factorisation.

- Clause 2 specifies that only those operations the specialisation clauses of which are coarsened should be refined. The reason for this is that the refinement should only be used to refine operations from which behaviour was removed (through the coarsening), in order to invoke the same behaviour through an intermediary operation. In the example this is correct as both the coarsening and refinement affect only $name_{11}$.

- Clause 3 specifies that the operation invocations that were removed through the coarsening should be added through the refining part of the connected extension. In other words, the intermediary operation should only have the behaviour that was removed from the original participant and nothing else. In the example, $name_{21}$ is removed from a specialisation clause through the coarsening and added to a specialisation clause through the connected extension.

- Furthermore, clause 4 requires the operation invocations added in clause 3 are the *only* operation invocations added to specialisation clauses through the connected extension.

- Finally, clause 5 ensures that the behaviour after the factorisation is the same as before. It specifies that the operation invocations added to an operation m through the refinement should refer to operations with the same behaviour as what m exhibited before the coarsening. In the example, $name_{11}$ now invokes $name_{31}$, which invokes $name_{21}$, the same operation $name_{11}$ invoked before.

### Impact on Conflicts

As always, the set of conflicts that can be caused through a factorisation is the union of the conflicts caused through its sub-operators. However, one case deserves some extra attention.

[2]Which is always possible, because the combined reuse modifiers can always be decomposed in their submodifiers.

**Operation Capture and Inconsistent Operations** Operation capture occurs when an operation invocation of m is added to a specialisation clause by one modifier and this same operation m is added or changed by another modifier. Similarly, inconsistent operations occur when an operation invocation of m is removed from a specialisation clause by one modifier and this same operation m is added or changed by another modifier. The first conflict thus appears after refinement, the second after coarsening. One would therefore expect both conflicts to appear after a factorisation. This is depicted in figure 4.7 in a simplified case, where there is only one participant and the factorisation thus only affects intra-participant behaviour. The notation $m\{n\}$ in the client interface is a short-hand notation for the same operation invocation attached to an acquaintance relationship of a participant with itself (i.e., intra-participant communication).

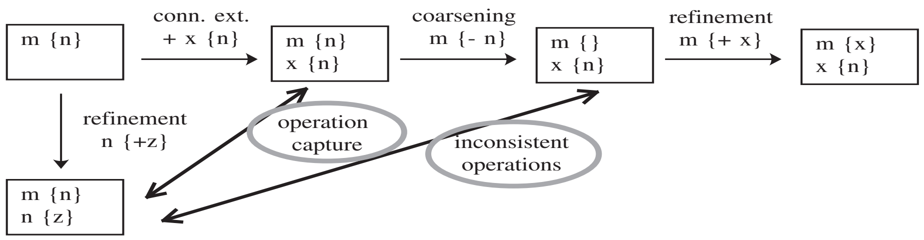


Figure 4.7: Annihilating Conflicts through Factorisation

Upon closer examination, it is obvious that there is an intuitive difference. When an operation invocation is removed from a specialisation clause through a factorisation, it is always added to another specialisation clause with the result that it remains within the transitive closure of the operation it was originally removed from. Therefore, one does not get true inconsistent operations. Therefore, we include the following rule.

**Rule 4.1 (Inconsistent Operations Annihilation)** The conflict of **inconsistent operations** that can be caused through a participant coarsening is **annihilated**, when this participant coarsening is part of a factorisation.

One could argue that because the conflict does not occur due to the fact that the removed invocation is still in the transitive closure, this conflict should be automatically annihilated when considering chains of modifiers as discussed in section 3.6. We feel however that annihilating all such coarsenings might also remove inconsistent operation conflicts that the user would like to be aware of. By explicitly declaring the coarsening to be part of the factorisation, however, the user explicitly expresses that the conflict can be neglected.

Note that operation capture is still possible, because if another modifier adapts one of the operations that was moved through the factorisation, this might still cause a problem.

**Overview** Table 4.3 gives an overview of the possible conflicts on factorisation. This table is again built by taking the union of the relevant parts of tables 3.1, 3.2 and 3.3 from chapter 3, only now we have left out inconsistent operations for the reasons explained above. Note that we do not have an extra column for refinement as all conflicts caused by refinement are already part of the set of conflicts caused by connected extension.

| | Factorisation | |
|---|---|---|
| | participant connected extension | participant coarsening |
| part. ext. | operation name, accidental operation capture, unanticipated recursion | - |
| cont. ext. | - | - |
| part. canc. | dangling operation reference | dangling operation reference |
| cont. canc. | dangling participant reference | dangling participant reference |
| part. ref. | operation invocation, unanticipated recursion, regular operation capture | operation invocation, regular operation capt. |
| cont. ref. | - | - |
| part. coars. | operation invocation, unanticipated recursion, regular operation capture | operation invocation |
| cont. coars. | dangling acquaintance reference | dangling acquaintance reference |

Table 4.3: Conflicts with Factorisation

## 4.6 Renaming

### Motivation

The final combined operator we introduce is renaming. This operator will usually be applied to give inferred participants a different name from the participant they were derived from. Even without changing a participant's interface one might at some point want to rename a participant. Therefore, we make this a distinct operator.

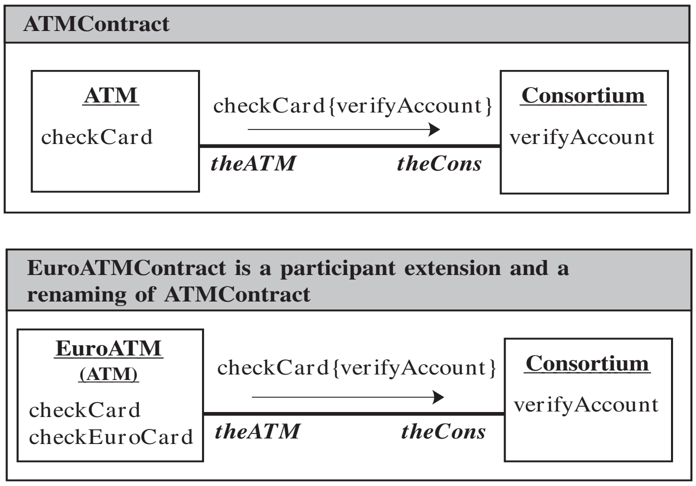


Figure 4.8: An Example of Renaming

Figure 4.8 shows a simultaneous renaming and participant extension. It adapts the `ATMContract` in order to handle ATMs that also handle Eurocards. While applying the participant extension only would result in a contract where the participant `ATM` still had the same name, here we also adapt the name. We adopt the convention to write the name of the participant that was renamed between braces below the new name, when this is useful as a clarification.

Renaming might seem to be a very straightforward operator, but it takes some work to define how it is built up from other modifiers. The reason for this is that we actually have to remove a participant with one name and then re-introduce a participant with another name and the exact same acquaintance clause and interface. We also have to change the names of all references to the renamed participant in acquaintance clauses on other participants.

We do not give a full definition here as that would lead us into too much tedious detail, but we use this operator (and the adopted notation) further on in examples.

## 4.7 Summary

We have shown how combined operators can be defined based on the basic operators of chapter 2 and defined some often recurring combinations. We did by no means give all possible useful combinations here, but it is clear from the examples that with the basic definitions of section 4.1 it is easy to define new combinations.

The conflicts that can be caused by these combined operators are most often the union of the conflicts that can be caused by the basic modifiers that they are made up from. It was shown in chapter 3 that conflicts can sometimes disappear when there are annihilating modifiers in a chain. It would of course make not much sense to take up such a sequence in a combined modifier. Still, sometimes conflicts can be ignored because of the meaning of the combined operator. Such an example was given with inconsistent operations caused by a participant coarsening that is part of a factorisation.