Chapter 3

Managing Evolution and Composition

In the previous chapter we introduced reuse contracts with their reuse operators. Reuse contracts provide structured documentation of software systems. We also claimed that the use of reuse operators makes it possible to detect the evolution conflicts we discussed in chapter 1. In this chapter we introduce rules for conflict detection based on reuse operators. Sections 3.1 to 3.5 give rules for situations where both modifications can be represented by one reuse modifier. Section 3.6 then discusses how this approach scales to modifications made by chains of modifiers. In the next two chapters we then elaborate on this basic model, while in chapter 6 we show how the rules presented in this chapter were used to manage the evolution of a small framework.

3.1 Evolution and Composition of Basic Modifiers

In this chapter, we discuss how the problems that were discussed in chapter 1 can be detected by means of reuse contracts and what the possible solutions for these problems can be. In order to do that we investigate how two modifications made to one reuse contract interact. This is depicted in figure 3.1. We call the original reuse contract which both operators will be applied to the *base (reuse) contract.* We name the two modifiers M_1 and M_2, the reuse contract corresponding to the application of M_1 to the base contract R_1 and the reuse contract corresponding to the application of M_2 to the base contract R_2. We then examine the effect of applying M_2 to R_1. In other words, we assume that M_1 and M_2 are two modifiers that are applicable to the base contract and we want to see whether the combination of both is possible and whether the result will be as expected. Note that although we need to give the modifiers names and therefore talk about M_1 and M_2, the order in which the modifiers are applied is irrelevant to the possible conflicts[1].

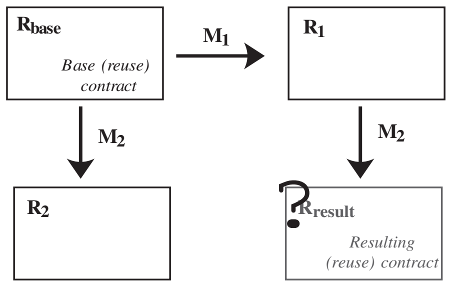

Figure 3.1: Base Reuse Contract Exchange

We follow the same classification of problems as in chapter 1. We first consider interface conflicts, then dangling reference conflicts and then conflicts concerning the calling structure. As we include information on the calling structure in the interfaces of reuse contracts and as all conflicts — except one — are detected by comparing the reuse modifiers, one could argue that all these conflicts are interface conflicts. Here, we make the distinction based on how a developer would perceive the different kinds of conflicts.

Interface and dangling reference conflicts are fairly basic and some of them can be detected by compilers of typed object-oriented languages. Interface conflicts occur when two operators introduce two items with the same name. Dangling reference conflicts occur when one operator removes an item from the interface, while the other operator refers to this item. This leads to inconsistent situations. Take as an example the case where one modification removes an operation, while the other modification adds an extra invocation of this same operation. In strongly typed languages such as C++ or Java this can be detected, in languages such as Smalltalk it will not be detected. The included information on the calling structure makes it possible to have interface conflicts on that level as well. These are currently not detected anywhere.

The conflicts concerning the calling structure we discuss in section 3.4 indicate much more serious problems, which now remain undetected in statically as well as dynamically typed languages. Conflicts such as operation capture, inconsistent operations and unanticipated recursion do not necessarily make a system break down, but may result in a working system that does not exhibit the expected behaviour: it does not behave the way the developers *assumed* it would. Conflicts such as the

[1]That is we will set up rules do detect conflicts when considering two independent modifiers. Of course, when looking at applicability definitions the order does play a role. This will be further discussed in section 3.5

ones with the counting set or the gateway and visitor packets in chapter 1 fall into this category.

We will see that there are different approaches possible to detect the conflicts.

- Most often it is possible to detect a conflict by comparing the modifiers;
- Sometimes a conflict may also be identified by looking at the applicability definitions;
- Sometimes, the information provided by the modifiers does not suffice and the base contract needs to be consulted or the resulting contract needs to be computed.

Where possible, we start by stating rules based on comparison of the modifiers. In section 3.5 we then discuss other options. In that section we also discuss the trade-off between being able to detect a maximum of conflicts and having flexible modifiers.

3.2 Interface Conflicts

The first category of conflicts concerns interface conflicts. These are caused by two modifiers adding the same kind of information to the base contract. Since there are four kinds of information in a reuse contract, there can be four kinds of interface conflicts. Since these four kinds of information are added by the four different basic operators the rules are straightforward. These conflicts always occur when the two modifiers represent the same operation adding or adapting items with the same name.

3.2.1 Operation Name Conflicts

We start with operation name conflicts. Operation name conflicts can occur when two modifications both introduce operations with the same name for the same participant. An example of an operation name conflict is depicted in figure 3.2, where both modifications introduce an operation `transaction` on the participant `Bank`. This situation needs to be signalled and either one of the two versions of transaction is preferred, or a combination of both needs to be made.

Since the introduction of new operation names is always achieved through participant extension, operation name conflicts can only occur when both modifications perform a participant extension (of the same participant).

Rule 3.1 (Operation Name Conflict) An **operation name conflict** occurs when an operation m and a participant p exist such that both M_1 and M_2 represent participant extensions by m on p .

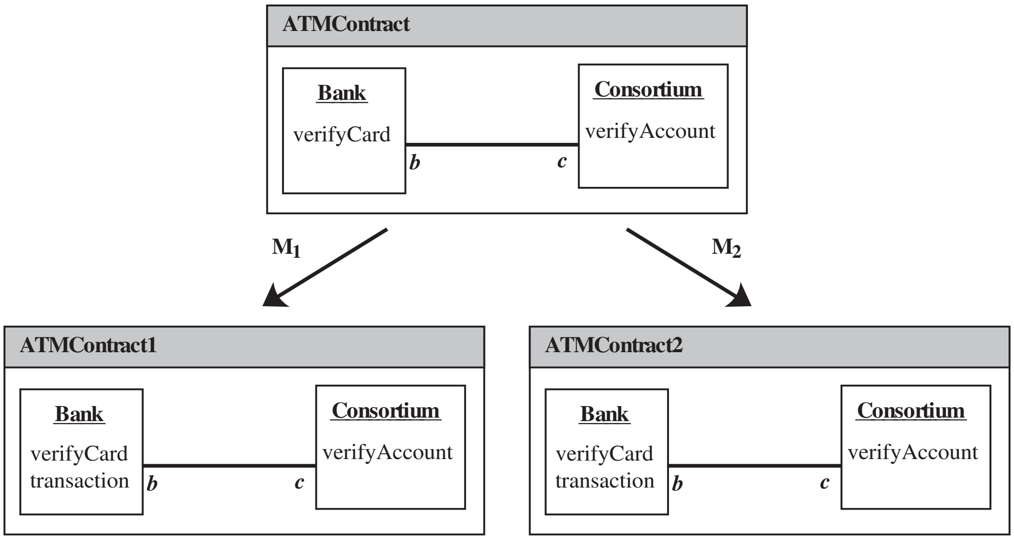

Figure 3.2: An Operation Name Conflict

Operation name conflicts can be resolved by either renaming one of the operations or by hiding one or both of these operations. Hiding is a technique that can be applied, for example, in object-oriented languages by declaring operations to be private to a class.

3.2.2 Participant Name Conflicts

A participant name conflict appears when both modifiers add a new participant with the same name. This can happen only through context extension.

Rule 3.2 (Participant Name Conflict) A **participant name conflict** occurs when a participant p exists such that both M_1 and M_2 represent context extensions by p .

The solution to this conflict can be either the renaming of one of the added participants or the merging of both participants. Whether the first or the second option is chosen will depend on whether both participants were introduced for similar purposes or not.

3.2.3 Operation Invocation Conflicts

The next two conflicts take a slightly different form. Before, a conflict was caused when two modifications introduced the same item. Here, a conflict occurs whenever two modifications alter the dependencies of the same item in a different way. Consider, for example, two modifications refining the same operation in a different way. We call this an operation invocation conflict. An example of such a conflict is given in figure 3.3. While the first modification refines the operation `transaction` to invoke `verifyAccount`, the second modification refines the same operation to invoke `processTrAct`. When keeping in mind that an implementation will be associated with both modifications, it is clear that combining both modifications leads to a problem. Picking one will neglect the refinement made by the other. Whether a combination of both is desirable depends on the situation. When it is, a combined refinement has to be established and the corresponding operation re-implemented.

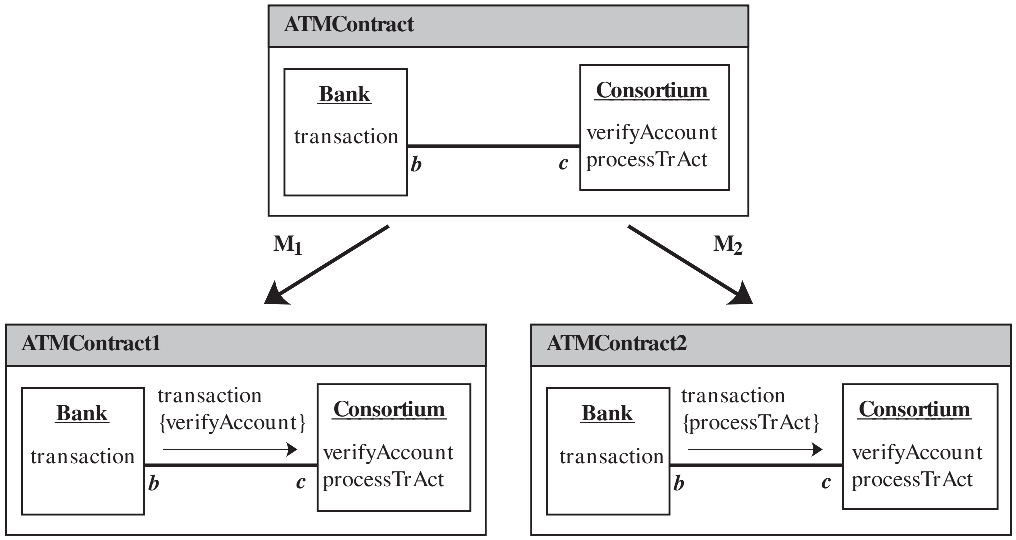

Figure 3.3: An Operation Invocation Conflict

Operation invocation conflicts can happen through participant refinement, but also through participant coarsening. We do not make a distinction between conflicts caused by refinement and conflict caused by coarsening here, because on a conceptual level the conflicts are completely similar. In both cases, two independent modifications change the specialisation clause of the same operation. The corresponding implementations will therefore not be compatible.

A conflict occurs when the specialisation clause of the adapted operation in M_2 does not represent a correct refinement or coarsening of the specialisation clause of the same operation in M_1. Note that this is the reason why we repeat the entire specialisation clause in the refinement operator. Otherwise we would not be able to detect — by looking at the modifier only — that the implementation attached to M_2 will not take into account the extra operation invocations added by M_1. It is in order to be able to express these conflicts clearly that we introduced the short-hand notations for repeating and resulting specialisation clauses in chapter 2. Recall that both participant refinements and coarsenings have to repeat the specialisation clauses on the original contracts of the operations they are adapting.

Rule 3.3 (Operation Invocation Conflict) An **operation invocation conflict** occurs when an operation m and a participant p exist, such that both M_1 and M_2 represent participant refinements or participant coarsenings of m on p and the repeating specialisation clause of m in M_2 is not identical to the resulting specialisation clause of m in M_1.

3.2.4 Acquaintance Relationship Conflicts

Acquaintance relationship conflicts are completely similar to operation invocation conflicts, but concern adding new acquaintance relationships to the acquaintance clauses. Acquaintance relationship conflicts can therefore only occur when both modifications perform a context refinement or a context coarsening. Therefore we also introduced short-hand notations for repeating and resulting acquaintance clauses.

Rule 3.4 (Acquaintance Relationship Conflict) An **acquaintance relationship conflict** occurs when a participant p exists, such that both M_1 and M_2 represent context refinements or context coarsenings of p , and the repeating acquaintance clause of p in M_2 is not identical to the resulting acquaintance clause of p in M_1.

Note again that to detect this conflict we repeat the acquaintance clause in the context refinement modifier. The solutions to this conflict are equivalent to the solutions of specialisation clause conflicts.

3.2.5 Summary of Interface Conflicts

To finish the discussion on interface conflicts table 3.1 states a summary of all conflicts. Note that we only completed one half of the table because it is symmetric. It is clear from the table that the operators are completely orthogonal: interface conflicts only occur due to interaction of two identical reuse operators. Note that we do not mention cancellation in this table. One could consider operation name or participant name conflicts to occur when both modifications remove the same item, but we chose not to consider this as a conflict here.

3.3 Dangling Reference Conflicts

The second category of conflicts are dangling reference conflicts. They occur when one operator removes an item from the interface, while another operator continues to refer to it. These conflicts can also be checked by means of rules comparing the modifiers.

	participant extension	context extension	participant ref./coars.	context ref./coars
participant extension	operation name	*no conflicts*	*no conflicts*	*no conflicts*
context extension	-	participant name	*no conflicts*	*no conflicts*
participant ref./coars	-	-	operation invocation	*no conflicts*
context ref./coars	-	-	-	acquaintance relationship

Table 3.1: Interface Conflicts

Note that there are three different dangling item conflicts while there were four kinds of interface conflicts. The reason for this is that from the four basic modelling constructs in reuse contracts (participants, acquaintances, operations and operation invocations), only the first three can be referred to by other constructs. For example, operation invocations refer to the participants the operations are defined on, to the operations themselves and to the acquaintance relationship along which the operation is invoked. Operation invocations are never referenced.

3.3.1 Dangling Operation Reference

Figure 3.4 depicts an example of a dangling operation reference. While the second modifier adds an invocation of `processTrAct`, the first modification removes exactly this operation. Combining the two modifications obviously leads to errors.

This conflict occurs when an operation is removed by M_1 and M_2 refers to this removed participant. This can only happen when M_1 is a participant cancellation and M_2 a participant cancellation, refinement or coarsening.

Rule 3.5 (Dangling Operation Conflict) A **dangling operation reference** occurs when an operation m and a participant name p exist such that M_1 represents a participant cancellation of m on p and M_2 a participant cancellation or refinement referencing m on p or a participant coarsening dereferencing m on p .

3.3.2 Dangling Participant Reference

This conflict occurs when one of the participants is removed by M_1 and M_2 refers to this removed participant. This can only happen when M_1 is a context cancellation. M_2 can be any possible operator except for context extension, because context extensions do not refer to existing participants.

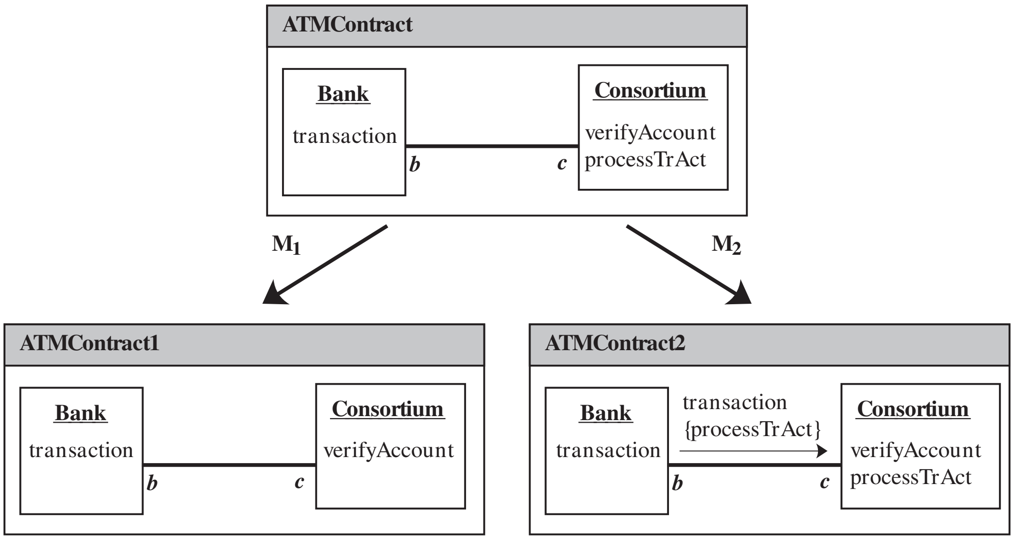

Figure 3.4: A Dangling Operation Reference

Rule 3.6 (Dangling Participant Conflict) A **dangling participant reference** occurs when a participant name p exists such that M_1 represents a context cancellation of p and M_2 is any modifier mentioning p except for a context extension modifier.

3.3.3 Dangling Acquaintance Reference

This conflict occurs when one of the acquaintance relationships is removed by M_1 and M_2 refers to this removed acquaintance relationship. This can only happen when M_1 is a context coarsening and M_2 is a participant extension, refinement or coarsening.

Rule 3.7 (Dangling Acquaintance Conflict) A **dangling acquaintance reference** occurs when an acquaintance name a exists such that M_1 represents a context coarsening of a on p and M_2 represents a participant extension, refinement or coarsening mentioning a on p .

3.3.4 Summary of Dangling Reference Conflicts

Note that for all these conflicts a distinction can be made between two cases. The first case is where M_1 adds information concerning the item removed by M_2, the second is where M_1 removes information concerning the item removed by M_2. For example, in the rule for dangling operation references, when an operation m is removed, we mention a conflict both when the second modifier references m and dereferences m. The first case constitutes a real problem. In the second case, this might not really be a conflict. We could have made distinct rules for these two cases for each of the previous conflicts, but did not want to include too many similar, basic conflicts. The only difference between the two cases is whether M_2 is an inverse operator or not.

Table 3.2 gives an overview of these conflicts. Note that the table does not list all possible operators. In order to keep the table concise, we grouped the operators by how they appear in the rules.

	context cancellation	participant cancellation	context coarsening
any operator except context extension	dangling participant reference	-	-
part. cancellation part. refinement part. coarsening	-	dangling operation reference	-
part. extension part. refinement part. coarsening	-	-	dangling acquaintance reference

Table 3.2: Dangling Reference Conflicts

3.4 Conflicts Concerning the Calling Structure

In addition to interface conflicts, we identified conflicts concerning the calling structure. In this section we set up rules to detect those conflicts by means of reuse contracts as well. We again accomplish this by investigating the modifiers. For the last conflict, unanticipated recursion, it is however not enough to consider only the two modifiers. The base reuse contract needs to be considered as well.

3.4.1 Operation Capture

But let us start with operation capture. Figure 3.5 gives an example of an operation capture. While the first modifier adds an invocation of `calcCode` by `verifyCard`, the second modifier adds an invocation of `verifyCard` by `verifyTrans`. The result of combining both combinations is that every time `verifyTrans` is invoked, this results in an invocation of `calcCode`. We say that `verifyCard` gets captured, because the changes made to it by the first modification have an influence on the second modification. This could amount to no problem at all, but it is also possible that this leads to unforeseen situations, or to an overhead. Maybe it is not necessary to invoke `calcCode` for every `verifyTrans`. Therefore, such situations should be signalled on integration. Note that any change to `verifyCard` made by M_1 would lead to operation capture in combination with M_2. Operation capture implies that a change to one operation might influence more operations than the person making the change expected.

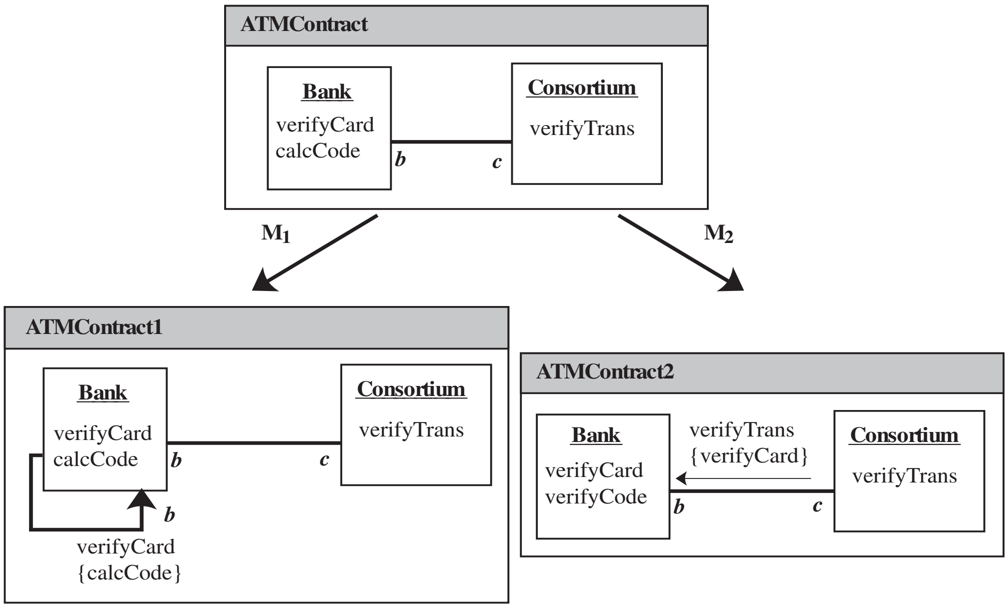

Figure 3.5: Regular Operation Capture

Remember that we distinguished regular operation capture from accidental operation capture. Figure 3.5 models a regular operation capture, because `verifyCard` was already present on the original reuse contract. We call it regular, because the person making the change to `verifyCard` knew that other modifiers could possibly add invocations to it.

Accidental operation capture would have occurred when `verifyCard` was not present on the original reuse contract and both modifications introduced it, while one added an invocation to it. We call this accidental because both developers introducing `verifyCard` could not foresee the other one introducing an operation with the same name. Note that accidental operation capture always coincides with an operation name conflict.

Regular Operation Capture

A regular operation capture occurs when an operation invocation of m is added to a specialisation clause by one modifier and this same operation m is changed by another modifier. As a consequence, operation capture can only be caused when one modifier is a refinement. The other modifier changes m, so it can only be a refinement or a coarsening.

Rule 3.8 (Regular Operation Capture) A **regular capture** of an operation n by an operation m occurs when

- M_1 represents a participant refinement of m on p referencing n on q ;
- M_2 represents a participant refinement of n on q or a participant coarsening of n on q .

The solution to regular operation capture depends on the situation. It is possible that the operation that is captured performs the desired behaviour, so there is no real conflict and nothing needs to be done. If not, the captured operation either has to be re-implemented to provide the desired behaviour or it has to be encapsulated so that it is no longer captured.

Accidental Operation Capture

If the operation m did not yet exist on the original base contract the operation capture is *accidental*, as one could not foresee it. Since this conflict can only occur when a dependency is added to a newly added operation, this can only occur after performing an extension.

This becomes apparent through the *operation name conflict* that always occurs simultaneously with the accidental operation capture, because the two modifications independently introduce the same operation.

Rule 3.9 (Accidental Operation Capture) An **accidental capture** of an operation n by an operation m occurs when

- M_1 represents a participant extension by m on p referencing n on q ;
- M_2 represents a participant extension by n on q ;

Note that in fact these two extensions also cause an operation name conflict. Note also that the first clause of this rule implies that n is also newly added, because of the self-containedness of extension. As a solution, both the name conflict and the capture have to be resolved in the ways described above.

3.4.2 Inconsistent Operations

While operation capture occurs when specialisation clauses are augmented, inconsistent operations appear when operations are removed from specialisation clauses. Consider the example in figure 3.6 where the first modification adds the invocation of `calcCode` by `verifyCard`, while the second modification removes the invocation of `verifyCard` by `verifyTrans`. The developer adding the invocation of `calcCode` assumed that this invocation would also have an effect on `verifyTrans`. After combining both modifications this is no longer the case. We say that `verifyTrans` and `verifyCard` have become inconsistent: one calculates codes, while the other does not. Whether this is a problem or not will again depend on the situation, more particularly on the reasons why the invocation of `verifyCard` by `verifyTrans` was removed. If this was for pure implementation reasons with as goal still exhibiting the same behaviour, there is a problem. To solve it the operation `verifyTrans` needs to be adapted in, to get their behaviour consistent again.

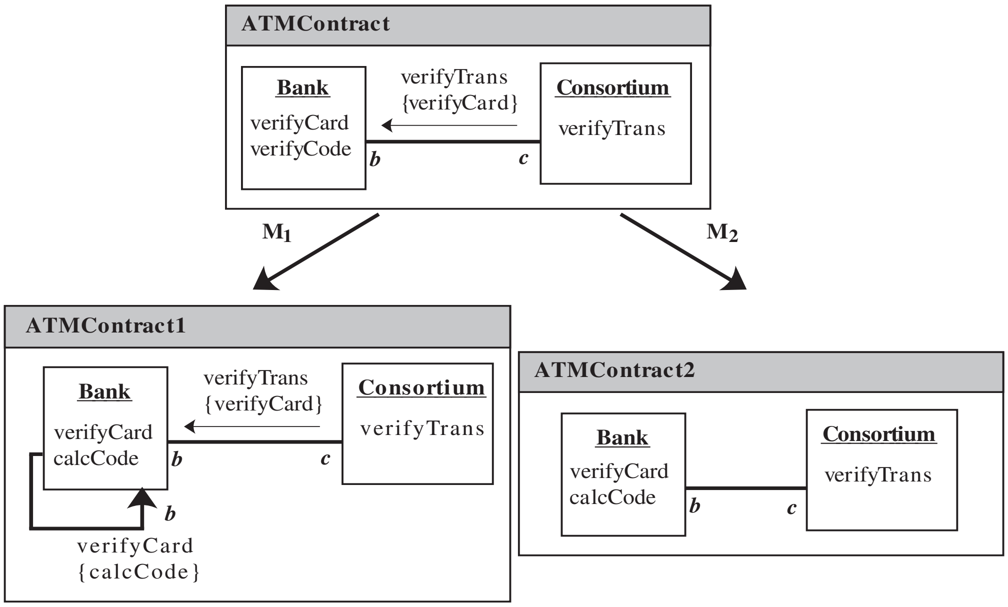

Figure 3.6: Inconsistent Operations

The removal of an operation from a specialisation clause can be achieved through coarsening and cancellation. However, when an entire operation is removed, it cannot become inconsistent with another operation. Therefore, this conflict only occurs after a coarsening. Again, the second modifier should change the operation that is removed from the specialisation clause, so it can only be a refinement or a coarsening.

Rule 3.10 (Inconsistent Operations) Two **operations** m and n become **inconsistent** when

- M_1 represents a participant coarsening of m on p dereferencing n on q ;
- M_2 represents a participant refinement or a participant coarsening of n on q .

To solve this problem the operation m needs to be adapted in M_1 in order to get their behaviour consistent again.

Note that this conflict was the one that occurred with the counting set and the gateway and visitor packets in chapter 1.

3.4.3 Unanticipated Recursion

Unanticipated recursion is the conflict that occurs when, after two separate augmentations of the specialisation clauses of two separate operations, these operations show mutually recursive behaviour. Take, for example, figure 3.7 where the first modification adds an invocation of `verifyAccount` by `verifyCard` and the second modification adds an invocation of `verifyCard` by `verifyAccount`

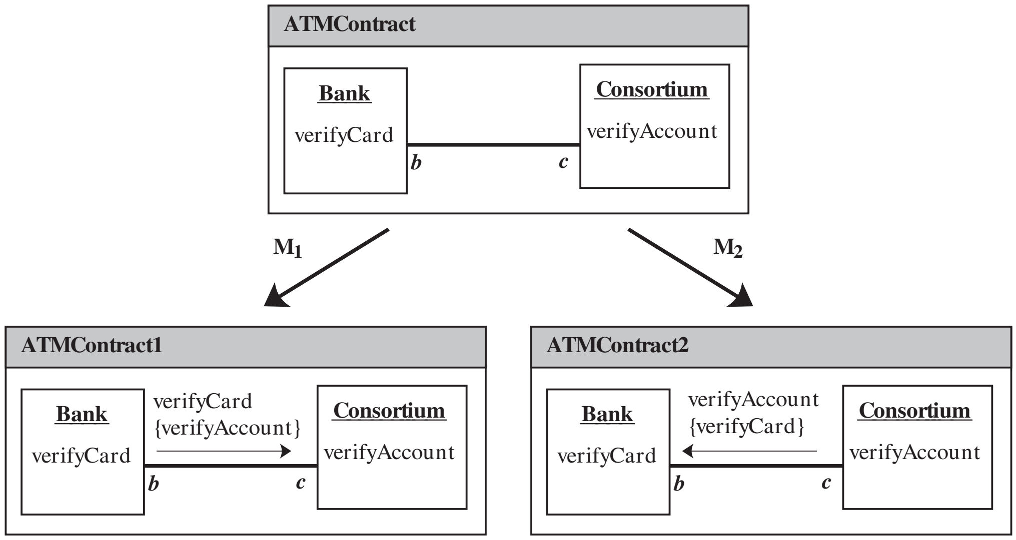

Figure 3.7: Unanticipated Recursion

A first straightforward version of the rule for unanticipated recursion is:

Rule 3.11 (Unanticipated Recursion) Unanticipated recursion of two operations m and m occurs when

- M_1 represents a participant refinement of m on p referencing n on q or a participant extension by m on p referencing n on q ;
- M_2 represents a participant refinement of n on q referencing m on p or a participant extension by n on q referencing m on p .

Note that because of the applicability rules the only combinations that can actually occur are two refinements or two extensions.

However, this version of the rule only detects the most simple cases of this conflict. Consider a more complex situation, as depicted in figure 3.8.

First we need to clarify our notation. What is important in the illustration of this and further examples is not the exact form of the modifiers, but the way the different modifiers interact. Therefore, now that the formats of the different modifiers have all been thoroughly discussed in chapter 2, we describe the modifiers in a more intuitive notation. We no longer depict the entire structure, but use a more intuitive notation (that is also used further on to describe examples and tools). In this new notation, we mark items that are added to the contract with a '+' and items that are removed from the contract with a '-'. When a complete interface or participant is added or removed, the '+' or '-' is placed in front of the description or the description's name. When something is changed in the specialisation or acquaintance clause only, the signs are placed there.

A second simplification, is that we denote intra-participant operations between braces in the interface of the participants, instead of along a loop. This allows us to give simple examples with contracts that only consist of one participant.

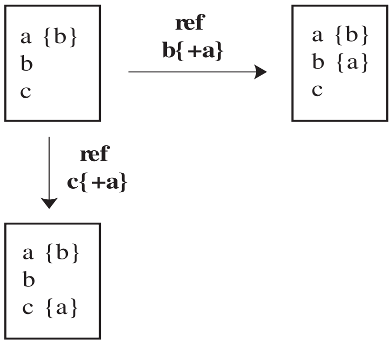

Figure 3.8: Indirect Unanticipated Recursion

Coming back to the case of unanticipated recursion, figure 3.8 depicts a situation where a participant in the original reuse contract contains three operations a, b, and c and a has specialisation clause containing b. If one modifier refines b to make an invocation of c, and the other modifier refines c to invoke a, when comparing both modifiers, there does not seem to be any problem, occurring to the rule given above. The global effect of both refinements is that an indirect invocation from b to a is introduced. However, when looking at the reuse contract this actually is a problem because a already invoked b and thus an unanticipated recursion occurs. So in general, not only both modifiers but also the specialisation interface of the original reuse contract needs to be taken into account. Furthermore, to be able to express this rule we need to introduce the notion of a transitive closure of a specialisation clause.

Definition 3.1 (Transitive Closure) The **transitive closure of a specialisation clause** SC is the union of SC and the transitive closure of specialisation clauses of all operations appearing in SC.

Based on this notion we can now give a more general rule for unanticipated recursion. When considering only the simplest case, this new definition boils down to the previous definition.

Rule 3.12 (Unanticipated Recursion Revisited) Unanticipated recursion of two operations m_1 on p_1 and m_2 on p_2 occurs when

- M_1 and M_2 are participant refinements (extensions) ;
- R_1 is a participant refinement (extension) of R with M_1, R_2 is a participant refinement (extension) of R with M_2, R_{21} is a participant refinement (extension) of R_2 with M_1;
- m_1 appears attached to an acquaintance name referring to p_1 in the transitive closure of the specialisation clause of m_2 in R_{21};
- m_2 appears attached to an acquaintance name referring to p_2 in the transitive closure of the specialisation clause of m_1 in R_{21};
- at least one of the last two statements was not true on R_1 and R_2.

Note that while for all other rules it sufficed to consider only the modifiers in order to detect the conflict, here we also need the original reuse contract, because we need to compute the results of the refinements or extensions.

3.4.4 Summary of Conflicts about the Calling Structure

To round off the discussion on conflicts involving the calling structure we give a summary of all conflicts in table 3.3. Again, only one half of the table needs to be considered, because the rules are symmetric.

3.5 Evaluation

3.5.1 Alternative Rules

We have tried to set up rules that were based only on comparison of the reuse modifiers. Except for unanticipated recursion, we succeeded in this goal. For some of the rules however a different approach is possible, namely for the interface and dangling reference conflicts. Take operation name conflicts as an example. Such a conflict occurs when both modifiers are participant extensions introducing an operation with the same name. Another way to detect this conflict is through applicability.

	part. extension	part. refinement	part. coarsening
part. extension	accidental operation capture, unanticipated recursion	*no conflicts*	*no conflicts*
part. refinement	-	regular operation capture, unanticipated recursion	regular operation capture, inconsistent operations
part. coarsening	-	-	inconsistent operations

Table 3.3: Conflicts concerning the Calling Structure

Rule 3.13 (Operation Name Conflict) An **operation name conflict** occurs when both M_1 and M_2 are participant extension modifiers, and M_2 is not applicable after M_1.

The correctness of this last rule can be checked by looking at the applicability definition and by taking into account that both M_1 and M_2 are applicable to the base reuse contract. The applicability definition contains four clauses only the second of which is dependent on possible intermediary participant extensions. Therefore, if M_2 is no longer applicable the second clause has failed. This clause states that the operation names in M_2 are different from all operation names already present on the referenced participant in the contract to which M_2 is applied. As this was valid for M_2 with respect to the base reuse contract, the only possibility is that M_1 added operations with names identical to operation descriptions in M_2. Again note that the order in which the modifiers are applied is irrelevant.

Dangling reference problems can also be detected by means of the applicability rules, but in this case the order in which the two modifiers are applied is important. This can be clarified when considering that all these conflicts are caused by the fact that one of the modifiers removes a certain item from the reuse contract that the other modifier in some way relies on. When the modifier that removes something is performed last, the conflict will not be detected through the applicability rules. An alternative rule for the dangling participant reference conflict is given below.

Rule 3.14 (Dangling Participant Reference) A **dangling participant reference** occurs when M_1 is a context cancellation modifier, and M_2 is not applicable to the exchanged base contract.

The conflicts regarding the calling structure cannot be detected merely by applicability rules, as we need more information about what exactly is part of the modifiers.

Note that in order to use this version of the rules, one needs to know the base reuse contract, because applicability of a modifier depends on the reuse contract one tries to apply it to. For the versions of the rules as given above, only the modifiers are necessary. We have therefore chosen for the first version because these rules are more general. When one knows the old reuse contract and the new one, it is easy to calculate the modifiers that were applied to make the adaptation. Knowing only the reuse modifiers does not give the same information on the base reuse contract.

3.5.2 Other Possible Conflicts

The set of conflicts that we have discussed above is obviously not complete. On the one hand, more conflicts can be conceived that could be detected by this basic version of reuse contracts, on the other hand the set of conflicts we can detect is limited by the information that we include in our reuse contracts. One could for example expect to find more conflicts concerning the acquaintance relationships. This is however restrained by the fact that we do not make a distinction between different kinds of acquaintance relationships, which makes it hard to say anything meaningful about acquaintance relationship conflicts. The basic reuse contract interface will therefore be extended in chapter 5.

On the other hand, the set of conflicts that can be detected is dependent on the information we include in the reuse modifiers. We did, for example, repeat the specialisation and acquaintance clauses of the base reuse contracts in the participant and context refinement modifiers in order to detect operation invocation and acquaintance relationship conflicts. By including this information we can detect extra conflicts but each modifier is applicable to fewer reuse contracts. A trade-off needs to be made in the definition of each reuse modifier.

Take the example of figure 3.9. The first modifier moves the acquaintance relationship with name `a` to point to `Z` instead of `Y`[2]. The second modifier adds an operation invocation along the acquaintance relationship with name `a`. Combining these two modifications causes the operation invocation to be performed on a different participant than might have been planned by the developer adding this invocation. Two questions occur. First, is this a conflict? Second, how can it be detected if it is?

The answer to the first question is hard to give at this level of abstraction. It depends on the operation invocation mechanism that is being modelled. The second answer is that if we want to detect this situation, it is not sufficient to specify the name of the acquaintance relationship along which an operation is invoked in

[2]Note that doing this implies first removing the original acquaintance relationship by means of a context coarsening and subsequently introducing the new one by means of a context refinement. We discuss combined operators in section 3.6 and chapter 4.

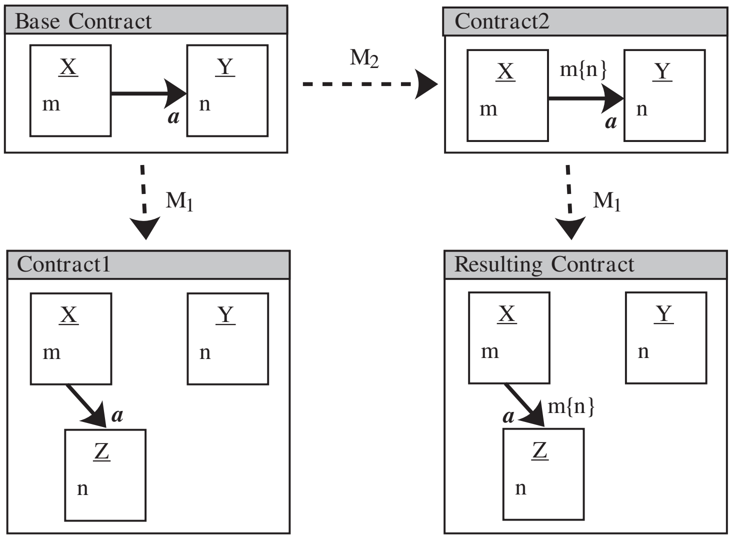

Figure 3.9: An Acquaintance Relationship Conflict

the participant refinement modifier. We also need to state to which participant this acquaintance relationship is pointing at the time of the refinement. Again, the information that is introduced to enable detecting another conflict restricts the number of reuse contracts to which a modifier is applicable. What information is to be included depends on the particular flavour of reuse contracts. Another option could be to let the user explicitly add any information he regards as being important. We run into this issue again when setting up reuse contracts for classes and objects in chapter 5.

3.6 Evolution of Chains of Adaptations

Until now, we have only discussed the problem of detecting conflicts when two basic modifiers are applied to the same reuse contract. The question remains which conflicts can or will occur when applying two *chains* of primitive modifiers to the same reuse contract. Due to the subsequent modifiers in a chain local conflicts may be annihilated. As modifications are usually modelled by a series of modifiers, this question is crucial to the usefulness of our approach. We give a sketch of how conflicts caused by chains of modifiers can be detected. To make the reasoning clearer this sketch only concerns the participant modifiers, but a similar argumentation can be made for the context modifiers and the interactions between both kinds of modifiers. Note that since we only use the participant versions of the operators. We omit the word “participant” when denoting the operators.

Our approach starts by first transforming the chains so that each modifier is independent — with respect to the possible conflicts — of the preceding ones, and then detecting conflicts by comparing the modifiers in both chains one by one. This approach works, except for conflicts concerning transitive closures. Even worse, a chain can introduce additional transitive conflicts.

To simplify the situation, we first investigate the case where a reuse contract is modified by a single primitive modifier on the one hand, and a chain of primitive modifiers on the other hand, as depicted in figure 3.10. Sections 3.6.1 to 3.6.3 discuss those conflicts that can be detected by comparing the modifiers only. Section 3.6.4 discusses transitive closure conflicts as well. Section 3.6.5 then summarises for the case of one modifier versus a chain of modifiers, while 3.6.6 generalises to two chains of modifiers.

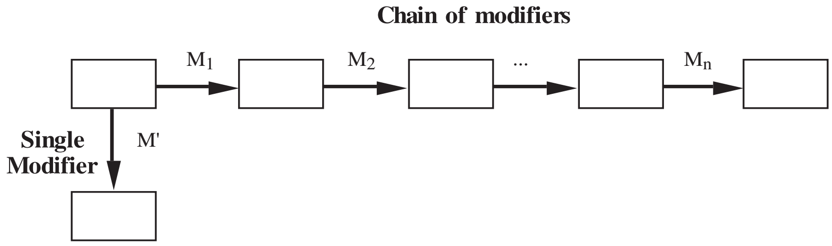

Figure 3.10: Chain vs. Single Modifier

3.6.1 Chain vs. Single Modifier

At first glance, we might expect that the possible conflicts that can occur are *exactly* those caused by the interaction of the single modifier and each of the modifiers in the chain. Two possibilities should be considered: the chain can cause *more* or *fewer* conflicts.

- *Is it possible that a chain causes more conflicts than the sum of those caused by the modifiers that are part of it?* When considering conflicts that can be detected by modifier comparison only, the answer is negative. Indeed, in that case, the chain is nothing more than a sequence, the conflicts of which can be detected by comparing each modifier in the chain with the single modifier. Apart from the fact that they need to be applied in a given order, no other assumptions are made about the relationship between the modifiers in a chain. So, the modifiers do not express additional assumptions about each other or the base reuse contract that can be broken and no new conflicts can arise.

- *Is it possible that a chain causes fewer conflicts than the sum of those caused by the modifiers that are part of it?* Here the answer is positive. It is possible that some conflicts that occur due to the interaction of the single modifier and one of the modifiers in the chain are *not* really conflicts when considering the chain as a whole. For example, when one of the modifiers in the chain introduces a new operation causing a name conflict, but the same operation is cancelled later on in the chain, there is *no* name conflict with respect to the chain as a whole.

In general, such annihilation of conflicts occurs only when some conflicting situation earlier on in the chain gets resolved due to a subsequent modification in the chain. Conflicts can only be annihilated by the *subsequent* modifiers. *preceding* modifiers have an impact on the applicability of later modifiers, not on the conflicts these modifiers cause.

In sections 3.6.2 and 3.6.3 we investigate the different combinations of modifiers and see which combinations can lead to annihilation of conflicts and which cannot.

3.6.2 Annihilation of Conflicts

Kinds of Annihilating Pairs

As argued above, some local conflicts in a chain should not be considered as conflicts when considering the chain as a whole. More precisely, local conflicts due to some modifier in a chain may be resolved by subsequent modifiers. We distinguish between two kinds of conflict annihilation.

The first case occurs when a modifier M_i in a chain modifies some operation m and a subsequent modifier M_j cancels m. Not only will the effect of what M_i did with the operation get lost, but also all possible conflicts in M_i with respect to that operation will vanish. Note that the cancellation M_j itself can give rise to a dangling operation reference, but all conflicts (operation capture, inconsistent operations, ...) due to the original modification of m, will be annihilated when m is cancelled. Note that in this case the annihilation is independent of the particular modification performed by M_i.

The second case is more subtle and concerns conflict annihilations where a modifier is followed by an inverse modifier annihilating part of its effect. An example is a refinement of an operation m with an operation invocation to n, followed by a coarsening of m dereferencing n. The refinement of m may introduce, for example, an operation capture of n, but this conflict disappears when the invocation of n is removed again by means of the coarsening[3]. Note that in this case the second modifier need not always be the direct inverse of the first modifier. It can, for example,

[3]One might wonder whether such situations will actually ever occur, or whether they are not just a sign of bad design. We will see later on that such situations can be found, for example, in inheritance hierarchies in languages with single inheritance only, in order to mimic multiple inheritance.

also be an extension, followed by a coarsening of an operation invocation introduced by this extension.

Elimination of Annihilations from Chain

By definition, the annihilated conflicts should not be signalled when detecting conflicts due to the interaction of a modifier and a chain of modifiers. Therefore, before starting conflict detection in a chain, we transform the chain such that it contains no more annihilating modifiers. Of course, we have to take care that the transformation removes these conflicts only, and no others. Below, we take a more detailed look at how such a transformation should be done. We distinguish between the different operator combinations that can lead to conflict annihilations.

1. **Extension by m followed by cancellation of m.** For every subchain starting with a modifier introducing a new operation m and ending with a cancellation of m (and no other extensions or cancellations of m in between), we try to eliminate this extension and cancellation. Because some modifiers in between may also depend on m we first need to eliminate these dependencies. Therefore, we first need to perform the steps 1a and 1b:

   (a) First, we eliminate all annihilating pairs of refinements and coarsenings of some operation n with m. How to do this is explained in steps 2 and 3. Second, if the extension by m itself added an operation n referencing m, we remove the possible annihilating pairs of the extension by m and a coarsening of n dereferencing m, as explained in step 6. After this, it can easily be shown that no more coarsenings or refinements of n with m remain:

   - No coarsenings of n with m remain, because this is only possible if the coarsening was preceded by a refinement of n with m or when the extension contained an operation n referencing m (these are the only two ways m could have been introduced in the specialisation clause of n). Both cases lead to annihilating pairs, and have just been worked away.
   - Also, no refinements of n with m remain: if they are followed by an annihilating coarsening they have been worked away; the other case is impossible because the cancellation of m would then not be applicable.

   (b) As all references to m in specialisation clauses have now been removed, we can simply discard all refinements and coarsenings of m. Afterwards, the subchain contains no more dependencies on m.

   (c) Finally, discard the extension by and cancellation of m. (Of course, if the extension by m is part of a larger extension modifier, we remove only the relevant part of this modifier. A similar remark holds for the cancellation and for the refinements and coarsenings in the previous step.)

2. **Refinement followed by coarsening of n referencing m.** Eliminate all annihilating pairs starting with a refinement of n with m, and ending with a coarsening of n with m, that have no other refinements or coarsenings of n with m in between. This can easily be done by removing both modifiers — or their relevant parts. Note that in this case the "relevant parts" can mean that only part of the specialisation clause in a refinement is removed. If a refinement adds more operations to n's specialisation clause than just m and these operation invocations are not removed, this part of the refinement must be preserved.

3. **Coarsening followed by refinement of n referencing m.** This case is completely analogous to the previous one.

4. **Cancellation of m followed by extension by m.** Next we eliminate all annihilating pairs starting with a cancellation of m, ending with an extension by m and no other cancellations of m or extensions with m in between. If there are other modifiers in between, this is not a problem, as these cannot depend on m. If the extension by m has the same specialisation clause as m immediately before the cancellation was performed, we can simply remove both the cancellation and the extension (or the relevant parts of the corresponding modifiers). Otherwise, we drop the cancellation and replace the extension by a refinement of m adding all extra operation invocations (with respect to the situation immediately before the cancellation) and/or a coarsening of m, removing all operation invocations m no longer performs.

5. **Refinement (or coarsening) of m followed by cancellation of m.** To resolve this case we simply remove the refinement (or coarsening) of m. We do not need to take any precautions as all references to m have been removed before the cancellation of m is performed (otherwise the cancellation would not be applicable), and thus it actually no longer matters whether the refinement (or coarsening) was performed or not.

6. **extension by m followed by coarsening of m.** The only case we have not considered yet is when an extension by an operation m referencing n is followed by a coarsening of the same operation m removing the invocation of n. But as every extension could conceptually be seen as a "pure" extension (i.e., with empty specialisation clauses) followed by a refinement, the argumentation here is essentially the same as in step 2.

We have now given a transformation algorithm to eliminate annihilating operations from a chain. Note that this algorithm was independent of the kinds of conflicts we want to detect. In other words, when other conflict situations are defined that we want to be able to detect, this part of the argumentation does not need to be repeated.

3.6.3 Dependence of Modifiers

In this section, we verify whether no other annihilating conflicts may remain after the above transformation. In other words, we need to investigate whether the remaining modifiers in the chain are "independent", in the sense that the conflicts caused by one modifier do not influence other modifiers in the chain.

Table 3.4 gives an overview of the possible relationships between modifiers in a chain. We imagine M_1 preceding M_2 in a chain. We then have three possibilities.

1. The two modifiers are *independent*. This occurs, for example, with two extensions. The fact that the two extensions appear in one chain implies that the one extension is applicable after the other. Therefore, they cannot contain the same items and so they are independent.

2. The two modifiers are *annihilating*. These are the cases discussed in the previous section.

3. The two modifiers are *dependent*. These are the cases we have not discussed yet: they are treated below.

First, note that a single combination of modifiers sometimes has multiple kinds of interactions in table 3.4. This depends on their contents. Take for example extension after cancellation. When these two modifiers affect the same operation (e.g., an extension by m and a cancellation of m) they are annihilating, when they concern different operators (e.g., an extension by m and a cancellation of n) they are independent. Therefore, the word independent should be marked in each compartment of the table. In order to keep the table concise, we have only mentioned "independent" when it is the only possibility.

M_1 / M_2	extension	refinement	cancellation	coarsening
extension	independent	independent	annihilating	independent
refinement	**dependent**	**dependent**	independent	**dependent**/ annihilating
cancellation	annihilating	annihilating	independent	annihilating
coarsening	**dependent**/ annihilating	**dependent**/ annihilating	independent	**dependent**

Table 3.4: Dependencies between Modifiers

Since independent modifiers will not influence the conflicts each of them causes and since we have treated the case "annihilating" above, all that is left to do is take a closer look at the remaining dependencies.

Extension versus Refinement/Coarsening Two of the remaining dependencies between modifiers are when M_1 is an extension, and M_2 is a refinement or coarsening. This does not necessarily mean that the conflicts of M_1 are influenced by M_2. Suppose M_1 is an extension by an operation m referencing n.

- If M_2 is a coarsening of m, it has no influence on the conflicts caused by M_1. The only possible conflicts for extension are operation name conflicts, dangling participant references, dangling acquaintance references and accidental operation captures. A coarsening clearly does not affect the occurrence of operation name conflicts, dangling participant references or dangling acquaintance references as it would therefore need to influence respectively the set of operations in an interface, the set of participants in a reuse contract and the set of acquaintance relationships in a contract[4]. Accidental capture of n by m could, however, be eliminated by a coarsening of m dereferencing n, but this case was already handled by the application of step 6 above.
- If M_2 is a refinement of m, this again does not influence possible operation name conflicts, dangling participant references, dangling acquaintance references or accidental operation captures occurring due to the application of M_1.
- If M_2 coarsens n or removes an invocation of m in some operation, this removes an indirect dependency between M_1 and M_2. If M_2 is a refinement of n or adds an invocation of m in some operation, an indirect dependency is created. The effect of such indirect dependencies is discussed in section 3.6.4, on transitive closure conflicts.

Refinements versus Coarsenings When M_1 and M_2 are refinements or coarsenings, indirect dependencies may also be created; for example, M_1 may introduce an invocation of an operation which is further refined by M_2. Again, we refer to 3.6.4. Apart from such situations, subsequent refinements or coarsenings have no influence on the conflicts caused by preceding refinements or coarsenings, unless they form an annihilating pair, but this case was discussed above.

3.6.4 Transitive Closure Conflicts

We saw in section 3.4.3 that some conflicts[5] — such as unanticipated recursion — cannot be detected by comparing modifiers one by one, but only by comparing the

[4] Recall that we consider *participant* coarsening only here.

[5] In addition to unanticipated recursion, other transitive closure conflicts are imaginable; we do not discuss them in this dissertation.

single modifier in combination with all preceding modifiers and the original reuse contract. In some degenerate cases it is sufficient to compare the two modifiers only. For example, if the one modifier refines an operation m with an invocation of n, and the other modifier does exactly the opposite, unanticipated recursion occurs. In more complex situations, where the recursion is not introduced straightforwardly, but through intermediate operations, not only both modifiers but also the specialisation interface of the original reuse contract needs to be taken into account. In general, this problem occurs whenever we need to take into account the transitive closure of specialisation clauses. Therefore, we call these conflicts "transitive closure conflicts".

When we are dealing with chains of modifiers, the situation becomes even worse. It is not sufficient to compare *and* the original reuse contract *and* the modifiers of both chains one by one. Indeed, the preceding modifiers in the chain are important as well. For example, if a refinement modifier adding an invocation by n of x is replaced by a chain of refinements n referencing a, a referencing b, b referencing c and c referencing x, the unanticipated recursion conflict still occurs.

Annihilation of Transitive Closure Conflicts

When we consider the entire chain, some of the transitive closure conflicts are not really conflicts either. Even when all annihilating pairs have been removed, it is still possible for inverse modifiers to appear in the chain. Because all annihilating pairs have been removed, the remaining inverse modifiers necessarily remove parts of the original contract. It is possible for these modifiers to annihilate transitive closure conflicts. Consider the following example, as depicted in figure 3.11.

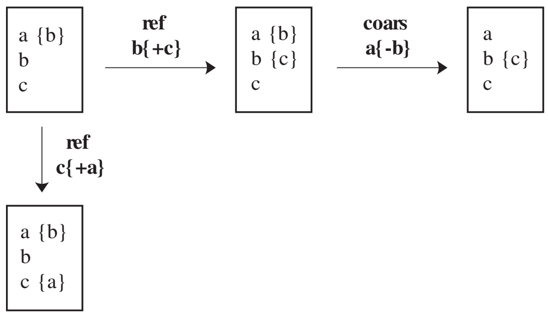

Figure 3.11: Transitive Closure Conflict Annihilation

A participant in a contract contains three operations a, b and c, of which a invokes b. In the chain a refinement is performed that causes b to invoke c. Later on in the chain a is coarsened. The single modifier refines c to invoke a. Without the coarsening, a case of unanticipated recursion would occur, but in this case it will not.

In order to identify situations where an unanticipated recursion conflict might have occurred somewhere down the chain but not in the final result, one must always consider the result of the entire chain and not the intermediate results. For example, the unanticipated recursion rule of section 3.4.3 states that when both modifiers are refinements or extensions, the result of applying *both modifiers* should be checked to see whether there are two operations in each other's transitive closure, that were not connected that way before both operators were applied. When considering chains the rule should be adapted such that when the single modifier is a refinement or an extension and refinements or extensions are present in the chain, the same should be verified in the result of applying *both the single modifier and the chain*. By taking this approach, we might have to compute the transitive closure of operations more often than strictly necessary. Maybe situations as in figure 3.11 could be detected with less computing work, but that would lead us too far for this sketch.

3.6.5 Summary: Single Modifier versus Chain of Modifiers

In order to detect all conflicts concerning a chain of modifiers versus a single modifier, the following steps need to be taken.

1. remove all annihilating modifiers;

2. for the conflicts that can be detected by investigating the modifiers only: check the remaining modifiers in the chain one by one against the single modifier;

3. for transitive closure conflicts: apply the rules on the result of applying both the chain and the single modifier.

This way we are certain that we have detected all conflicts.

3.6.6 Conflicts between Two Chains of Modifiers

Now that we have discussed how to handle one modifier versus a chain of modifiers, it is not so hard to see what to do when considering two chains of modifiers, as in figure 3.12.

Again, first all annihilating modifiers need to be solved, now in each of the chains. When this is done, all modifiers from the two chains need to be checked, one by one. Again, the only conflicts we have missed there are transitive closure conflicts. To detect these, we now need to compute the result of both chains and apply the rules for transitive closure conflicts considering both results.

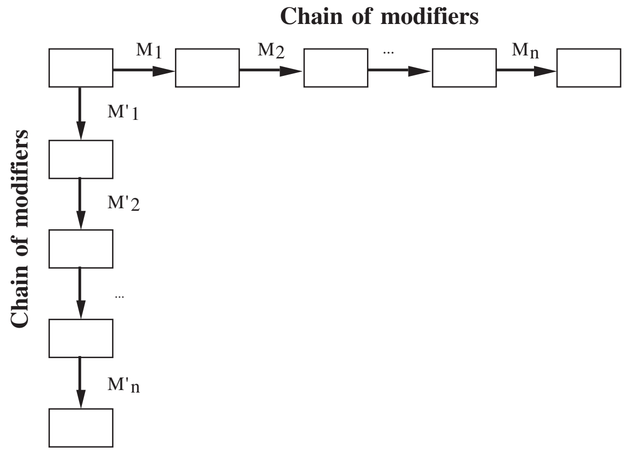

Figure 3.12: Two Chains of Modifiers

3.6.7 Conclusion

We have presented a sketch of how the conflict detection rules for single modifiers can be extended to chains of modifiers. A large part of this sketch, i.e., the removal of annihilating modifiers, is independent of the conflicts we want to detect. Only the second part as discussed in section 3.6.3 (i.e., the part on dependent modifiers), was dependent on the kind of conflicts we are concerned with. This implies that when we want to be able to detect other conflict situations, only this second part of the argumentation needs to be repeated.

We only gave a construction of how this can be handled for the participant modifiers, but a similar reasoning can be developed for the context modifiers and the interactions between both kinds of modifiers.

The possibility to use the rules for chains of modifiers as well is crucial, because the basic modifiers as defined in chapter 2 are very rudimentary and do not suffice to model real evolutions and variations in components. In the next chapter, we discuss some combinations of basic modifiers that occur very often and that we therefore want to define and name explicitly.