Chapter 2

Basic Reuse Contracts

2.1 Definition of Reuse Contracts

This dissertation introduces reuse contracts as a solution to the problems with incremental development of reusable components as discussed in the previous chapter. In this chapter we give a basic definition of reuse contracts and their operators. In general, reuse contracts consist of two major parts. First, they hold an extended interface description, providing information about the internal structure of a component which is necessary for reusers. Second, the reuse operators provide structured information on how different versions or parts of reusable components relate to each other.

We therefore first define interface descriptions and then a set of basic operators. Along with the definition we introduce a graphical notation, which will be used throughout the dissertation in most examples. Chapters 3 and 4 discuss how this basic model can be used to manage change propagation. In chapter 5 we then apply this basic definition to object-oriented systems. We do not immediately define reuse contracts for object-oriented systems, because we believe that the basic definition in this chapter is applicable to a broader range of components than just classes and objects. We come back to this scalability issue in chapter 8.

2.1.1 Participants

A basic reuse contract consists of a number of related *participants*. Each participant has a name, an acquaintance clause and an interface.

Definition 2.1 (Reuse Contract) A **reuse contract** is a set of participants, each with

1. a name that is unique within this reuse contract;

2. an acquaintance clause;

3. an interface.

Acquaintance clauses and interfaces are further discussed in the following sections.

2.1.2 Acquaintance Clauses

In order to be able to work together the participants in a contract have to be acquainted in some way. We say that there is an acquaintance relationship between the participants[1]. Acquaintance relationships are graphically depicted by a thick line between the two related participants, called a binding, as in figure 2.1. Throughout this chapter we use the example of an automatic teller machine (ATM) to illustrate our notation and definitions. It is a simple example, but sufficient to illustrate our ideas. More elaborate and real-life examples will be provided in later chapters.

In figure 2.1 we see two participants with names `ATM` and `Consortium` (representing a consortium of banks that share automatic teller machines). The line between the two participants represents two acquaintance relationships: one of `ATM` with `Consortium` and one of `Consortium` with `ATM`. When no arrows are drawn at the ends of the line, as in the picture, there are two acquaintance relationships. When depicting just one acquaintance relationship, an arrow is added to the line, pointing from a participant to the participant it is acquainted with. That is if `ATM` was acquainted with `Consortium` and not vice versa, we would have an arrow pointing from left to right in figure 2.1.

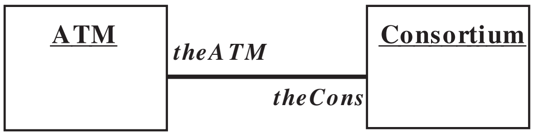

Figure 2.1: Two Acquainted Participants

An acquaintance relationship has a name. This name is noted along the binding, on the side of the participant that the acquaintance relationship points to. In the figure, `ATM` has an acquaintance `theCons` that points to the participant `Consortium`. It might seem awkward to place the acquaintance name on that side of the binding, but this corresponds to the notation of most methodologies. The rationale behind this choice is that the acquaintance name can also describe the role a participant plays in an acquaintance relationship.

We can now give the definition of an acquaintance clause.

[1]For now we make abstraction of what an acquaintance relationship corresponds with in actual code. An acquaintance relationship could be anything ranging from an association or a parameter binding to the transitive closure of a series of acquaintance relationships.

Definition 2.2 (Acquaintance Clause) An **acquaintance clause** is a set of acquaintance relationships $a.p$, associating an acquaintance name a with a participant name p .

The acquaintance clause of `ATM` contains only one acquaintance relationship: `theCons.Consortium`. The acquaintance clause of `Consortium` also contains exactly one acquaintance relationship: `theATM.ATM`.

Note the difference between the uses of the words 'acquaintance' and 'participant'. All reuse contracts are composed of a number of participants and every participant is possibly acquainted with a number of other participants, called its acquaintances. As a short-hand for the definitions of the operators we introduce the following notation.

Notation 2.1 When the acquaintance clause of a participant p contains $a.q$, we say that a on p **refers to** q or simply that p **refers** to q .

Further on, we also use the following terminology.

Definition 2.3 (Context) The set of participants in a contract together with the acquaintance relationships between them is called the **context** of a contract.

2.1.3 Client Interface

In order to be able to perform the behaviour represented by the reuse contract each participant has a number of operations defined on it. The set of all operation names defined on a participant is called this participant's *client interface.* Client interfaces are depicted graphically as in figure 2.2. Here `ATM` has two operations defined on it: `checkCard` and `transactionRequest`.

Figure 2.2: A Participant's Client Interface

We still need to further define the form of interfaces.

Definition 2.4 (Interface) An **interface** is a set of operations each consisting of

1. an operation name that is unique within this interface,
2. a specialisation clause.

The set of operation names of an interface constitutes the client interface.

Note that we only use operation names and not signatures. This definition can later be extended with operation annotations as abstract and concrete, but this will not affect the uniqueness of the operation names. To handle typing and overloading, the entire signature can be used as a unique name.

2.1.4 The Specialisation Interface

Extra information about internal dependencies between operations is described in what is called the *specialisation interface*. A specialisation interface is a collection of the different specialisation clauses, which are attached to operations. The specialisation clause of an operation enumerates the operations on which that operation relies[2].

We graphically represent specialisation clauses by means of operation names attached to the acquaintance relationships. The annotation `checkCard{verifyAccount}` along the binding from `ATM` to `Consortium`, as depicted in figure 2.3, signifies that the operation `verifyAccount` is invoked on `Consortium` by the operation `checkCard` of `ATM`. We call this an *operation invocation*. As this binding is bi-directional (i.e., it represents two acquaintance relationships), a thin arrow is added to show the direction of the operation invocation. When the binding is uni-directional the arrow can be omitted.

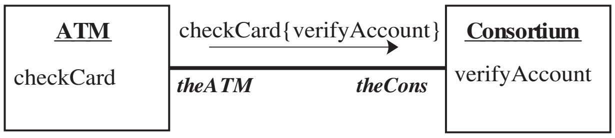

Figure 2.3: Part of the Protocol between Two Participants

Note that it is not allowed to put just any operation invocation anywhere. The operations that are referred to have to exist on the concerning participants and an acquaintance relationship in the right direction must be present.

For the annotation $x\{y\}$ along an arrow from participant1 to participant2 to be correct the following conditions need to be fulfilled:

- x has to be an element of the client interface of participant1 and y an element of the client interface of participant2;

[2]This terminology is based on [Lam93], where John Lamping first introduced specialisation interfaces. That paper focused on specialising classes through inheritance and on documenting self sends. One could argue for the use of the term reuse interface or composition interface in our case, but we use specialisation interface, because it is more general and to acknowledge the origins of our ideas.

- participant1 refers to participant2.

When an operation x on participant1 invokes more than one operation on participant2, this is denoted by an enumeration as in $x\{y, \ldots, z\}$. Note that it is also possible for participant1 to be equal to participant2. A participant can be acquainted with itself. This kind of binding can be graphically represented by a loop. Operations along such a loop represent intra-participant behaviour.

Finally we define specialisation clauses. Recall that each specialisation clause is part of each operation in an interface as stated in definition 2.4.

Definition 2.5 (Specialisation Clause) A **specialisation clause** is a set of operation invocations $a.m$, associating an acquaintance name a with an operation name m .

The specialisation clause of `checkCard` on `ATM` in figure 2.3 has one element: `theCons.verifyAccount`. Again as a short-hand for the definitions of the operators we introduce the following notation.

Notation 2.2 When the specialisation clause of an operation m on a participant p contains $a.n$, and a on p refers to q we say that m (on p) **invokes** n (on q).

A remark must be made about the kind of information that specialisation interfaces provide. In our case, specialisation interfaces are specified by listing operation dependencies purely based on operation names. One can also include type information, or other semantic information about, for example, the order in which operations must be invoked. Basic reuse contracts only indicate which operations rely on which other operations. The inclusion of extra information is to be seen as an extension of the basic model.

Moreover, a declaration that one operation m relies on another operation n does not mean that m will invoke n every time. n can, for example, be within a conditional in m. The fact that n is in m's specialisation clause only means that n might get invoked by m and therefore, there is a dependency between them. This makes the compliance of code with reuse contracts checkable, without the need for data flow analysis. We only need to consider static information.

2.1.5 The ATM Example

Figure 2.4 depicts a basic example of a reuse contract. The rectangle is used to denote the borders of a contract, while the shaded area allows to give the contract a name. This shaded area can also be used to denote from which other reuse contract a contract is derived and how (i.e., by means of which operator).

To summarise, figure 2.4 represents a reuse contract containing two *participants* `ATM` and `Consortium`. Each participant has an acquaintance clause and an

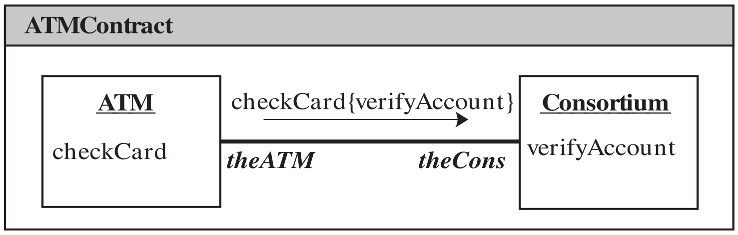

Figure 2.4: The ATM Reuse Contract

interface. The *acquaintance clause* of `ATM` contains the acquaintance relationship `theCons.Consortium`, the acquaintance clause of `Consortium` contains the acquaintance relationship `theATM.ATM`. The *interface* of `ATM` contains one *operation* with *name* `checkCard` and a *specialisation clause* containing the *operation invocation* `theCons.verifyAccount`. The interface of `Consortium` contains one operation with name `verifyAccount` and an empty specialisation clause. We say the `ATM` *refers to* `Consortium` and `Consortium` refers to `ATM`. We also say that `checkCard` (on `ATM`) *invokes* `verifyAccount` (on `Consortium`).

The combination of the definitions above gives a strict definition of reuse contracts. As reuse contracts are defined as sets of tuples, the definitions of the operators on reuse contracts can be expressed by means of unions, intersections and differences on these sets. Such a representation is the basis for the automated checking of rules on change propagation, as well as all kinds of tools using reuse contracts. The `ATMContract` is then mathematically denoted as follows.

```
ATMContract =
{ ( ATM,
    { theCons.Consortium },
    { (checkCard, {theCons.verifyAccount}) }
  ),

  ( Consortium,
    { theATM.ATM },
    { (verifyAccount, {})}
  )
}
```

This example clearly demonstrates the structure of reuse contracts. A reuse contract is a set of participants, which are triples consisting of a name, an acquaintance clause and an interface. In our example, there are two such triples, with respectively the names `ATM` and `Consortium`. An acquaintance clause is a set of acquaintance relationships, each relating an acquaintance name to a participant name. In our example, both acquaintance clauses are singletons. For example, `ATM` has an acquaintance name `theCons` associated with the participant name `Consortium`.

Each interface is a set of operations, which are pairs that consist of a name and a specialisation clause. In the example both participants have only one operation defined on them. `ATM` has an operation named `checkCard` and `Consortium` has an operation named `verifyAccount`.

A specialisation clause is a set of operation invocations, each relating an acquaintance name with an operation name. For example, the specialisation clause of `checkCard` on `ATM` contains an invocation associating acquaintance name `theCons` with operation name `verifyAccount`. The specialisation clause of `verifyAccount` on `Consortium` is empty. Note that as reuse contracts, interfaces, acquaintance clauses and specialisation interfaces are all sets, each of them can be empty.

2.1.6 Well-Formedness

Because we do not want any dangling references in reuse contracts, for example, an operation invocation of an operation that is not present in the contract, reuse contracts need to comply with some well-formedness conditions:

Definition 2.6 (Well-Formedness) A reuse contract R is **well-formed** if for every participant p the following conditions hold:

1. for each acquaintance relationship $a.q$ in the acquaintance clause of p : a participant with name q exists in R (WF1);

2. for each operation invocation $a.m$ in a specialisation clause in p :

   (a) a is an acquaintance name in the acquaintance clause of p (WF2);

   (b) m is the name of an operation in the interface of the participant a refers to (WF3).

All conditions can be checked on our example.

- The first restriction says that all participants that are named in acquaintance clauses must be part of the reuse contract. This is the case in our example as the acquaintance clause of `ATM` refers to `Consortium` and vice versa.

- The second restriction says that a participant can only invoke operations on participants it is acquainted to. This is correct in the example as `checkCard` only invokes an operation along `theCons`, which is part of ATM's acquaintance clause.

- The third restriction says that only those operations can be invoked on a participant that are part of its interface. In other words, no unexisting operations are invoked. In the example, `verifyAccount` is invoked on `theCons`, which refers to the participant `Consortium`, which has an operation `verifyAccount` in its client interface.

Note that the second and the third condition were already mentioned on page 44, when introducing the graphical notation. The first condition is automatically respected in the graphical notation as a line representing a binding can only be drawn to another participant of the contract.

2.2 Operators on Reuse Contracts

The extended interface description provided by reuse contracts is only a first step towards solving the problems discussed in chapter 1. Without information on the calling structure, i.e. which operations rely on which other operations, it is difficult to detect problems such as operation capture and inconsistent operators. Reuse contracts assist in the detection of such problems by making specialisation interfaces explicitly available. But even making specialisation interfaces explicit does not suffice in order to detect problems on evolution and composition. More information is needed both on the assumptions that can be made by reusers about the components and on the way the components are actually reused. Without this information, conflicts can only be detected by meticulously comparing the reuse contracts of the component that was reused, the new version of the component and the reusers. This is neither practical (in practice the original component might not even be available anymore), nor intuitively compelling.

We propose a methodology that is more intuitive for both reusers and developers of reusable components and which guides them in managing changes to these components. It is based on a classification of the possible changes that can be made to a reuse contract. Basically, a reuse contract has four modelling elements: participants and acquaintance relationships, which together form the *context*, and operations and operation invocations. Each of these elements can be added or removed from a contract, which leads to eight basic operators[3] as depicted in table 2.1.

Both at participant and at context level an *extension* adds the most basic element: respectively, operations and participants. Then, again at both levels, *refinement* introduces new relationships between these basic elements. Participant refinement connects operations by adding invocations, context refinement connects participants by adding acquaintance relationships. *Cancellation* and *coarsening* are the inverses of extension and refinement, respectively. Together these operators can model all possible adaptations to a reuse contract. In this chapter we define the different basic operators, in the next chapter their interactions are examined and it is discussed how this information can be used to detect and solve conflicts. In chapter 4 we define more complex operators based on combinations of these basic

[3]Note the difference: we have eight reuse opera*tors*, while interfaces contain opera*tions*.

Operator Name	Meaning
Participant Extension	adding new operations
Participant Cancellation	removing operations
Participant Refinement	adding new operation invocations
Participant Coarsening	removing operation invocations
Context Extension	adding new participants
Context Cancellation	removing participants
Context Refinement	adding new acquaintance relationships
Context Coarsening	removing acquaintance relationships

Table 2.1: Basic Operators

ones.

As reuse operators describe how a reuse contract is derived from another one, these operators are described by means of modifiers[4]. In general, a reuse modifier can be defined as follows.

Definition 2.7 (Reuse Modifier) A **reuse modifier** consists of a modifier tag and a modifier description.

A specific reuse modifier needs to be defined for each operator. The modifier tag corresponds to the name of the operator the modifier wants to model, e.g., "context extension". The modifier description describes the form of the modifier. Depending on which operator a modifier is modelling different information is necessary. For example, in order to model a context refinement the modifier contains participant names and acquaintance relationships between them. The modifier descriptions of an operator and its inverse operator do however always have the same form. To some extent information about the reuse contract a modifier is applied to needs to be repeated. For example, when performing a participant extension it is not sufficient to give the new operations, it is also necessary to identify the participants these operations must be added to. Also information that at first sight might seem redundant is sometimes added to the modifier. This is done in order to be able to detect more conflicts. There is always a trade-off between the amount of information that is added to a reuse modifier and its flexibility. The more detailed information is added, the more conflicts are detectable, but the less flexible the modifier becomes. This trade-off is discussed in more detail after we have discussed conflict detection.

Each operator is defined by means of three definitions and one property:

- *A modifier definition:* this definition describes the form of a modifier modelling this particular operator and some requirements it needs to fulfil. These

[4]We use the term operator as an intuitive name, while the modifier actually describes the difference between two reuse contracts and is formally defined.

requirements are independent of the reuse contract the modifier will be applied to;

- *An applicability definition:* this definition describes what properties a modifier must comply with in order to be applicable to a particular reuse contract. In other words, these are the requirements that are dependent of the reuse contract the modifier will be applied to;

- *An operator definition:* this definition describes how the result of applying this kind of modifier is determined;

- *A well-formedness property:* each application of a reuse operator to a well-formed reuse contract results in another well-formed reuse contract. In order to achieve this, some requirements are incorporated in the modifier and applicability definitions[5].

In all definitions below, R and R_x represent reuse contracts, while M and M_x represent modifiers. The subscripts of R and M represent the operator, for example, M_{pe} represents a participant extension modifier, while R_{pe} represents the result of applying participant extension. In the examples we do not explicitly mention the modifiers when they are clear from the context. We start each section introducing an operator with a motivation that is given using the ATM example. After that, we give the definitions and property, followed by a graphical illustration of the definition. Finally, some short-hand notations that shall prove helpful when discussing change propagation are given.

Readers that do not want a thorough understanding of each of the definitions, but just want to know what kind of operators there are and what they look like can skip the actual definitions. The examples and illustrations of the definitions suffice to get a working knowledge.

2.2.1 Participant Extension

Motivation

The goal of participant extension is to add new operations to one or more participants in a contract. This operator is usually applied in order to add new functionality to a participant and thus to the contract. Figure 2.5 shows a participant extension of the ATM contract, where the operations `trActRequest` (transactionRequest) and `processTrAct` (processTransaction) are added.

Extension is self-contained: the newly added operations refer in no way to the existing operations. They can, however, refer to each other as is shown in the example,

[5]Another approach would have been to omit these requirements from these definitions and then describe as a different property which requirements an operator must comply with in order to preserve well-formedness. We opted for the former, because we never desire ill-formed reuse contracts, not even as intermediary results.

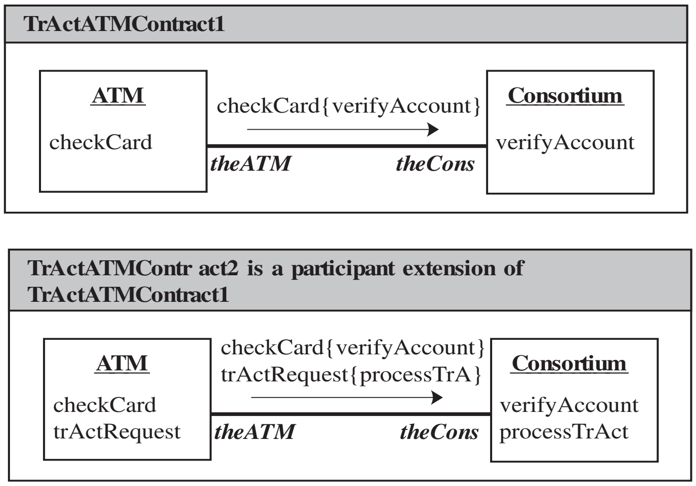

Figure 2.5: An Example Participant Extension

where `transactionRequest` invokes `processTransaction`. As a self-contained extension is completely independent of the reuse contract it is extending, this operator might not be desired so often in practice, but we want to keep the basic operators as orthogonal as possible. We will show later how participant extension can be combined with participant refinement to add dependencies between the operations added through the extension and the already existing operations.

Definition and Properties

Participant extension of contracts is defined in terms of a participant extension modifier. Recall that we define each operator in three steps: the modifier, the applicability and the operator.

Definition 2.8 (Participant Extension Modifier) A **participant extension modifier** is a reuse modifier with modifier tag "participant extension" and a modifier description which is a set of pairs (p, int) each consisting of a participant name p and an interface int [6].

Definition 2.9 (Participant Extendible) A reuse contract R is **participant extendible** by a participant extension modifier M_{pe} if for each pair (p, int) in M_{pe}[7]:

1. p is a participant name in R;

[6] Recall that an interface is a set of operations each consisting of a name and a specialisation clause.

[7] Note that the pairs (p, int) are actually part of the modifier description in M_{pe}, but we immediately say "each pair (p, int) in M_{pe}" to keep the definitions concise.

2. no operation name in int appears in the interface of participant p in R;

3. for each operation invocation $a.m$ in a specialisation clause in int :

   (a) a is an acquaintance name in the acquaintance clause of p in R;

   (b) if a on p refers to q then m is an operation in the interface of q in M_{pe}.

Note that the last clause is almost an exact copy of *WF2* and *WF3*. They are necessary to ensure that the result of a participant extension is again a well-formed reuse contract. The repetition of these clauses occurs in a number of applicability definitions. Here a slight adaptation was necessary, because of the self-containedness of extensions. The operation m referred to in an invocation also needs to be part of the modifier M_{pe} and not of R. However, to which participant an acquaintance name refers can only be seen in R.

Definition 2.10 (Participant Extension) If a reuse contract R is participant extendible by a modifier M_{pe}, then the reuse contract R_{pe} is the **participant extension** of R by M_{pe}, where:

1. R_{pe} contains all participants of R that are not mentioned in M_{pe};

2. for each (p, int) in M_{pe}: R_{pe} contains a participant with the same name and acquaintance clause as p in R and that contains all operations of p , plus int .

In the resulting contract all the participants of the original contract that are not mentioned in any pair (p, int) are maintained (clause 1). Furthermore, all participants of the original contract that are mentioned in a pair (p,int) keep their original name and acquaintance clause, while their interface is extended with the operations from int.

Note that the fact that R is participant extendible by M_{pe} implies that M_{pe} is a participant extension modifier. Note also that for a given reuse contract and a given participant extension modifier there is exactly one reuse contract that is the result of the participant extension of this reuse contract by this modifier. The above definition described how it is determined.

The following property can be proven about these definitions.

Property 2.1 *A participant extension of a well-formed reuse contract is well-formed.*

Proof The well-formedness definition (see page 47) imposes 3 constraints.

- WF1 concerns only the acquaintance clauses. As these are not altered through participant extension and the base reuse contract was well-formed, this condition also holds on the resulting reuse contract.

- WF2 states that operation invocations only occur between participants that are acquainted. This is ensured by the clause 3a in the participant extendibility definition.

- WF3 states that every operation that is invoked must be part of the client interface of the participant it is invoked on. This is ensured by clause 3b in the participant extendibility definition, which states that all operations in specialisation clauses in M_{pe} must be part of M_{pe}, attached to the required participant name. As the operations that already existed on R remain unchanged and R was well-formed, the property is also preserved for the operations that already existed in R.

Illustration

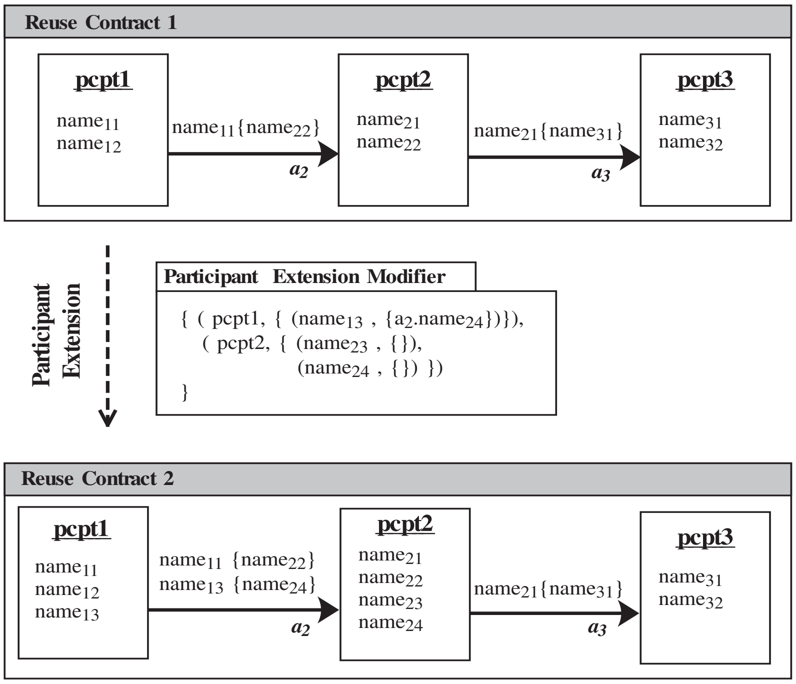

Figure 2.6: Participant Extension

Figure 2.6 illustrates a participant extension. As definition 2.8 states, a participant extension modifier contains pairs each consisting of a participant name and an interface. In the figure, there are two such pairs, one for pcpt1 and one for pcpt2. The name of the participant is necessary to know to which participant the operations are added; the associated operations are those to be added to this participant. In the example, one operation, $name_{13}$, is added to pcpt1 and two operations, $name_{23}$ and $name_{24}$, are added to pcpt2.

The extra constraints given in definition 2.9 on participant extendibility can also be checked on the example.

1. The first constraints says that all participant names occurring in the modifier must also occur in the original reuse contract. This is logical, as the participant names specify to which participant an operation must be added.

2. The second constraint states that only operations with new names can be added. The reason for this is obvious, as operation names are unique on every participant and we want to model the addition of new functionality, not the alteration of existing functionality, which is done through other operations.

3. Constraint 3a is the repetition of WF2, and is necessary to ensure well-formedness of the resulting reuse contract. It states that a participant can only invoke operations on its acquaintances. Note that in the example an operation invocation from pcpt2 to pcpt1 would have been illegal, as the binding between them is uni-directional.

4. Constraint 3b corresponds to WF3. It confirms that, after the participant extension, operations that are invoked on certain participants are part of that participant's interface.

   Furthermore, it ensures that the extension is self-contained. All newly added operations refer only to other newly added ones in their specialisation clauses.

   The example respects these constraints, as $name_{13}$ only invokes $name_{24}$, which is part of the participant associated to a_2, and the two operations added to pcpt2 have an empty specialisation clause.

Definition 2.10 determines the result of a participant extension.

1. All participants of the original contract, that are not mentioned in the modifier remain exactly the same. In the example, pcpt3 is unchanged.

2. The participants that are named in the modifier are completely preserved (including their name and acquaintance clause), except for the addition to their interface of the operations that are attached to them in the modifier. In the example, $name_{13}$ is added to the interface of pcpt1 and $name_{23}$ and $name_{24}$ are added to the interface of pcpt2. Note that the operations are copied including their specialisation clauses. This can be seen by the extra operation invocation that is depicted along the binding between pcpt1 and pcpt2.

Short-Hand Notations

In the rules on change propagation in chapter 3 we need not only the definitions as given above. Sometimes we also need information on the contents of a modifier. For example, an operation name conflict occurs when two participant extensions add an operation with the same name to the same participant. It is therefore not enough to know that two participant extension modifiers are applied, it is also necessary to know what operations are added by the extension and to what participants. For some conflicts it is also necessary to know what operation invocations are in their specialisation clauses. Because we want to take abstraction of the form of the modifier descriptions in the conflict detection rules, we introduce some extra definitions as a form of syntactic sugar to every operator definition. In most cases, the second is an elaboration on the first. For example, here the first definition states which operation is added to which participant, while the second definition also includes part of this operation's specialisation clause.

Notation 2.3 A participant extension modifier M_{pe} represents a **participant extension by** m **on** p if M_{pe} contains a pair (p, int) with an operation m in int

Notation 2.4 A participant extension modifier M_{pe} represents a **participant extension by** m **on** p **referencing** n **on** q if M_{pe} contains a pair (p, int) with an operation m in int , with m invoking n on q

When using these definitions, or the analogous definitions that are defined for other operators, it is always possible to leave out some part of the information. For example, when it is irrelevant to which participant an operation is added we simply say "an extension by m".

2.2.2 Context Extension

Motivation

Besides adding operations to existing participants, one can also introduce entirely new participants. Therefore, we introduce context extension. A context extension of the ATMContract is depicted in figure 2.7.

Again, context extension is self-contained. A context extension adds new participants to a contract, but the acquaintance clauses of these new participants can only refer to names of participants that are added through the same extension. In the example, `Bank` could have had an acquaintance relationship with yet another newly added participant, but not with `ATM` or `Consortium`. In fact, a context extension modifier is an entire reuse contract (in the example, a reuse contract with one participant), that is disjoint from the reuse contract it is extending. Adding dependencies between these two parts is achieved through context refinement.

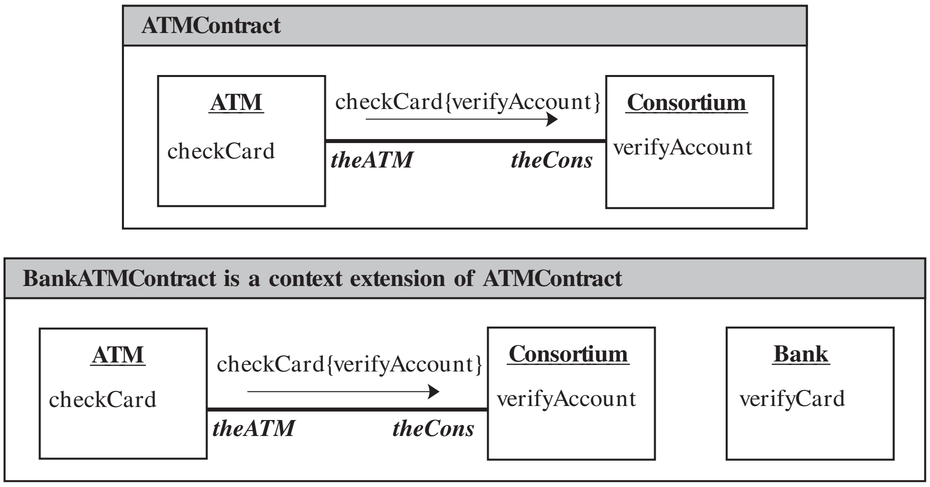

Figure 2.7: An Example Context Extension

Definition and Properties

Definition 2.11 (Context Extension Modifier) A **context extension modifier** is a reuse modifier with modifier tag "context extension" and a modifier description which is a well-formed reuse contract R_{Mod}.

Because the extension and the original part are disjoint, the extendibility definition becomes straightforward.

Definition 2.12 (Context Extendible) The **context** of a reuse contract R is **extendible** by a context extension modifier M_{ce} with modifier description R_{Mod} if for each participant p in R_{Mod}:
p 's name is different from all participant names in R;

Context extension of contracts can then be defined as follows.

Definition 2.13 (Context Extension) If a reuse contract R is context extendible by a modifier M_{ce} with modifier description R_{Mod}, then the reuse contract R_{ce} is the **context extension** of R by M_{ce} where:
R_{ce} contains all participants of R and all participants of R_{Mod}.

Property 2.2 *A context extension of a well-formed reuse contract is well-formed.*

Proof As both the added reuse contract and the original reuse contract are well-formed and the two parts do not influence each other, well-formedness is preserved after the extension.

Illustration

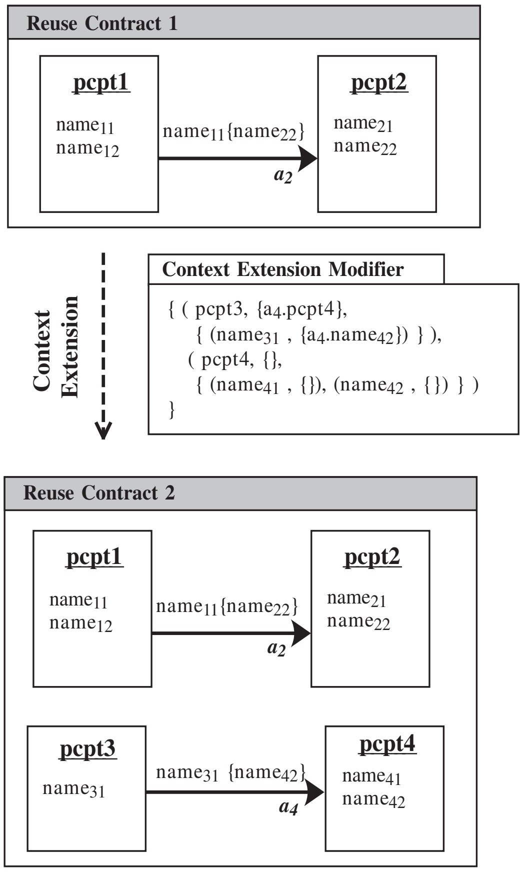

Figure 2.8: Context Extension

Figure 2.8 illustrates a context extension. A context extension modifier contains a well-formed reuse contract, i.e. a set of participants, that is completely disjoint from the original contract. The result of a context extension is a reuse contract that contains two disjoint parts: the contract that was extended and the contract it was extended with.

Note that a software system will always be described by a collection of contracts, not just one. As context extensions are completely disjoint from the rest of the contract, instead of performing a context extension, the contract could be added to the 'system' as a completely new contract. When this is not desirable, dependencies between the extended part and the original part can be introduced by means of refinements.

Short-Hand Notations

Again as a form of syntactic sugar to make the change propagation rules more readable we introduce some extra definitions.

Notation 2.5 A context extension modifier M_{ce} represents a **context extension by** p if a participant p is mentioned in M_{ce}.

Notation 2.6 A context extension modifier M_{ce} represents a **context extension by** p **referencing** q if participants p and q exist such that p and q are mentioned in M_{ce} and p refers to q .

2.2.3 Participant Cancellation

The opposite of extension is cancellation. We again first discuss this operator at participant level.

Motivation

As with extensions, cancellations are self-contained. This means that when an operation is cancelled, all operations it refers to in its specialisation clause need to be cancelled as well. When one wants to cancel an operation, but not the operations it refers to, first a coarsening (removal of operation invocations from the specialisation clause, see section 2.2.7) has to be performed[8].

On the other hand, operations can only be removed if they are not referred to by other operations, as that would violate the well-formedness of reuse contracts.

As an example of participant cancellation we can return to the first two reuse contracts we presented, but now in the reverse order. This is depicted in figure 2.9. While in section 2.2.1 we said `TrActATMContract` is a participant extension of `ATMContract`, here we can say `ATMContract` is a participant cancellation of `TrActATMContract`.

Definition and Properties

As we mentioned in the introduction of reuse operators, the modifiers of inverse operators always take the same form as the original operator's modifier. Therefore a participant cancellation modifier takes the same form as a participant extension modifier.

Definition 2.14 (Participant Cancellation Modifier) A **participant cancellation modifier** is a reuse modifier with modifier tag "participant cancellation" and a modifier description which is a set of pairs (p, int) each consisting of a participant name p and an interface int .

[8]Note that while we discuss how a developer should model systems here, in later chapters we will discuss how tools can be developed that assist the developer in doing this. Developers do not need to perform the more cumbersome tasks as described above manually.

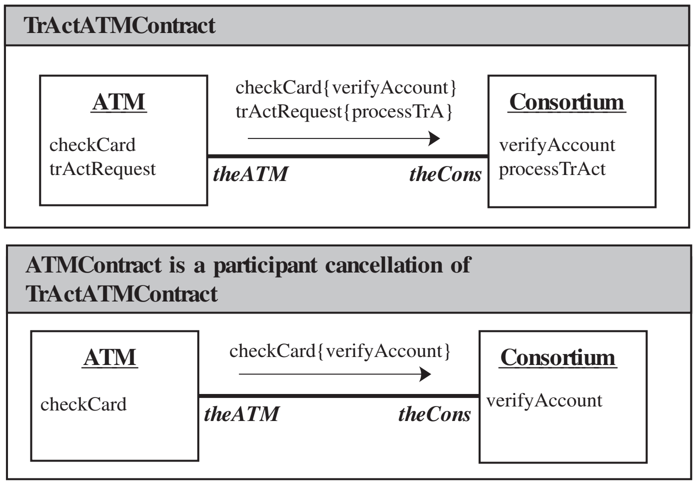

Figure 2.9: An Example Participant Cancellation

In order to be applicable the modifier needs to comply with the following constraints.

Definition 2.15 (Participant Cancellable) A reuse contract R is **participant cancellable** by a participant cancellation modifier M_{pc} if for each pair (p, int) in M_{pc}:

1. p is a participant name in R and each operation in int is identical to an operation in this participant in R ;
2. for all operations m , n and for all participants q in R , such that m on q invokes n on p : if n is an element of int , then m appears associated with q in M_{pc}.

Note that the last constraint is necessary to ensure that no operations are removed that are referred to by other operations. Again this is necessary to preserve well-formedness.

Definition 2.16 (Participant Cancellation) If a reuse contract R is participant cancellable by a modifier M_{pc}, then the reuse contract R_{pc} is the **participant cancellation** of R by M_{pc}, where:

1. R_{pc} contains all participants of R that are not named in M_{pc};
2. for each (p, int) in M_{pc}: R_{pc} contains a participant with the same name and acquaintance clause as p in R and that contains all operations of p , except for those in int .

Property 2.3 *A participant cancellation of a well-formed reuse contract is well-formed.*

Proof The well-formedness definition imposes 3 constraints.

- WF1 and WF2 are not affected by applying participant cancellation as they only concern the context;
- the second clause in the participant cancellability definition ensures that WF3 is respected by checking that only operations are removed that are not invoked by any other operations, except by operations that are themselves removed by the same cancellation.

Illustration

Figure 2.10 illustrates a participant cancellation. The modifier contains participant names with sets of operations attached to them. In the resulting reuse contract these operations are removed from the participants they are attached to. Note that $name_{31}$ can only be removed because $name_{21}$ — that refers to it — is removed by the same cancellation.

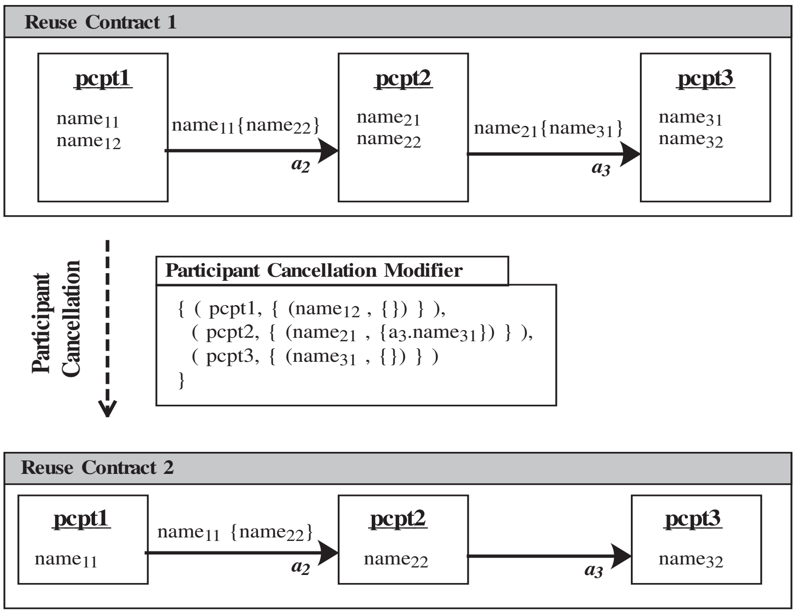

Figure 2.10: Participant Cancellation

Note that it might seem redundant to repeat the specialisation clauses of the operations that are to be removed. We do repeat this information in order to detect particular conflicts and to preserve the identity between the forms of modifiers of inverse operators.

Short-Hand Notations

Notation 2.7 A participant cancellation modifier M_{pc} represents a **participant cancellation of** m **on** p if M_{pe} contains a pair (p, int) with an operation m in int .

Notation 2.8 A participant cancellation modifier M_{pc} represents a **participant cancellation of** m **on** p **referencing** n **on** q if M_{pe} contains a pair (p, int) with an operation m in int , with m invoking n on q .

2.2.4 Context Cancellation

Context cancellation is the opposite of context extension.

Motivation

Again, context cancellation is self-contained, but we need to verify that no participants are removed that are referred to by other participants.

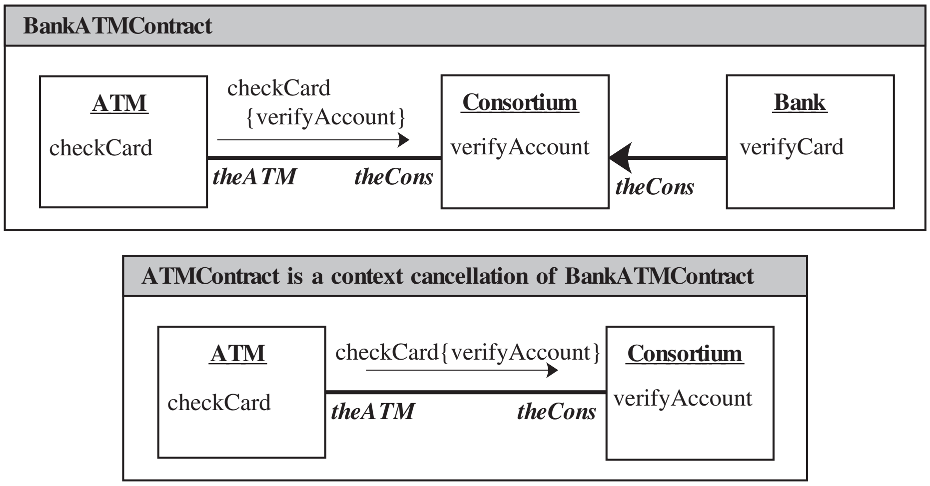

Figure 2.11: An Example Context Cancellation

In figure 2.11 `ATMContract` is only a correct context cancellation of `BankATMContract` because the binding between `Bank` and `Consortium` is uni-directional. If `Consortium` also had an acquaintance name pointing to `Bank`, removing `Bank` would have lead to an ill-formed reuse contract. We will show later that, when necessary, context coarsenings can be applied to first remove the superfluous acquaintance relationships.

Note that both `ATM` and `Bank` have an acquaintance relationship named `theCons` that refers to `Consortium`. This is not a problem, because acquaintance names only need to be unique on the participants they are defined on, in this example `ATM` and `Bank`. We will often encounter examples, where different acquaintance relationships referring to the same participant have the same name.

Definition and Properties

Again, the modifier of a context cancellation has the same form as the context extension modifier, i.e., it contains a complete and well-formed reuse contract.

Definition 2.17 (Context Cancellation Modifier) A **context cancellation modifier** is a reuse modifier with modifier tag "context cancellation" and a modifier description which is a well-formed reuse contract R_{Mod}.

Again, applicability becomes straightforward. We need to check that the participants that are to be removed are actually part of R and that no participants are removed while still referred to.

Definition 2.18 (Context Cancellable) The **context** of a reuse contract R is **cancellable** by a context cancellation modifier M_{cc} with modifier description R_{Mod} if for each participant p in R_{Mod}:

1. p is identical to a participant in R ;
2. p does not appear in the acquaintance clause of a participant in R that is not in R_{Mod}.

Context cancellation of contracts can then be defined as follows.

Definition 2.19 (Context Cancellation) If a reuse contract R is context cancellable by a modifier M_{cc} then the reuse contract R_{cc} is the **context cancellation** of R by M_{cc} where:
R_{cc} contains all participants of R , except for those named in M_{cc}.

We again prove that the operator preserves well-formedness.

Property 2.4 *A context cancellation of a well-formed reuse contract is well-formed.*

Proof

- WF1 is respected in view of the constraint in the second clause of the context cancellability that only participant names are mentioned (and thus removed), that are not referred to by other participants;
- Because the original reuse contract was well-formed and because only participants that are not referred to are removed, the names of the removed participants appear in no specialisation clause of R_{cc} and thus WF2 and WF3 cannot be broken.

Illustration

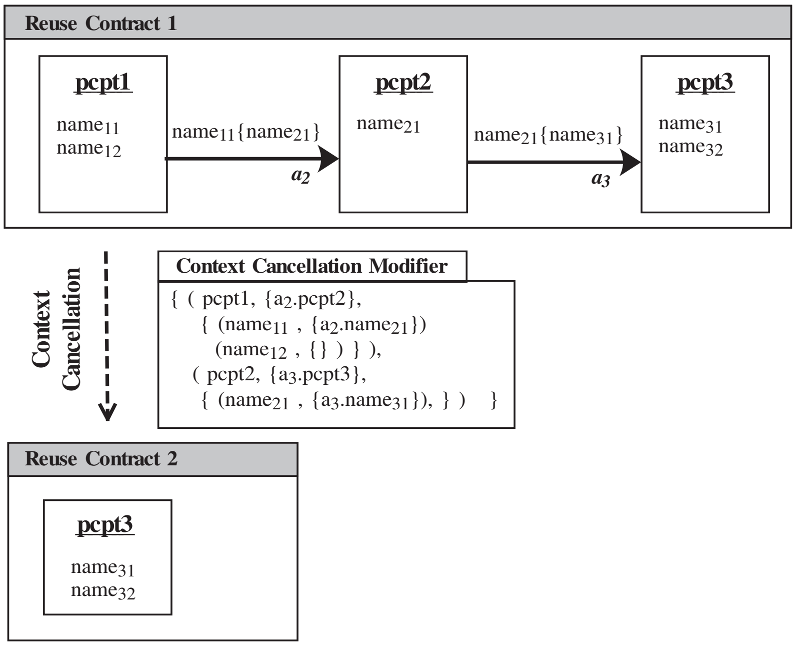

Figure 2.12: Context Cancellation

Figure 2.12 illustrates a context cancellation. The modifier contains a well-formed reuse contract containing two participants, `pcpt1` and `pcpt2`. They are identical in the modifier and in `Reuse Contract 1`. In the resulting contract the participants that are named in the modifier are removed. Note that `pcpt2` can only be removed because `pcpt1`, that refers to it, is removed as well. Participants can only be removed if they are not referred to or if the participants that refer to them are removed as well.

Short-Hand Notations

Again as a form of syntactic sugar to make the rules more readable we introduce some extra definitions.

Notation 2.9 A context cancellation modifier M_{cc} represents a **context cancellation of** p if a participant p is mentioned M_{cc}.

Notation 2.10 A context cancellation modifier M_{cc} represents a **context cancellation of** p **referencing** q if participants p and q exist such that p is mentioned in M_{cc} and p refers to q .

2.2.5 Participant Refinement

Motivation

In the previous four sections we have introduced extensions and cancellations, i.e., operators that add or remove elements from a reuse contract. In the next four sections we introduce refinements and coarsenings, i.e., operators that add or remove dependencies between elements of the reuse contract. Again, these operators act both at the participant and the context level, adding or removing respectively operation invocations and acquaintance relationships.

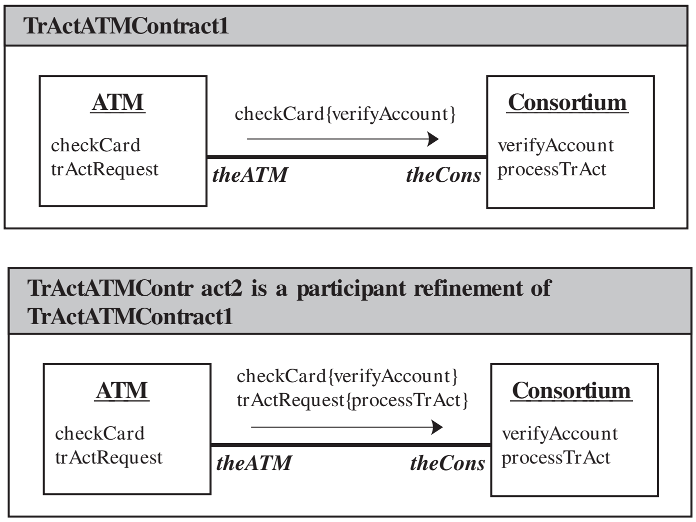

Figure 2.13: An Example Participant Refinement

We start with participant refinement. This operator adds extra operation invocations to the contract. Figure 2.13 shows a participant refinement: an operation invocation of `processTrAct` by `trActRequest` is added.

Note that a participant refinement can only add invocations between existing operations. It cannot add new operations. This needs to be done through participant extension.

Definition and Properties

A participant refinement adds extra operation invocations to the specialisation clauses of particular operations. In the reuse modifier we also repeat the original specialisation clause of this operation on the reuse contract it is refining. This is necessary to detect particular conflicts, for example, when two independent modifications refine the same operation. This issue is further discussed in chapter 3.

It is not enough to verify that the new specialisation clause is a superset of the original one. We need to explicitly state in the modifier which parts of the specialisation clause are newly added and which are repeated. Therefore, we introduce extended interfaces. An *extended interface* is an interface which contains operations with *two* specialisation clauses each. Incorporating this definition we can define participant refinability. We start again with the modifier.

Definition 2.20 (Participant Refinement Modifier) A **participant refinement modifier** is a reuse modifier with modifier tag "participant refinement" and a modifier description containing pairs $(p, extint)$ each consisting of a participant name p and an extended interface $extint$.
An extended interface is a set of operations, each consisting of an operation name and 2 disjoint specialisation clauses.

The disjointness of the two specialisation clauses is important, as the first repeats the specialisation clause of the base reuse contract, while the second describes the operation invocations that need to be added.

Definition 2.21 (Participant Refinable) A reuse contract R is **participant refinable** by a participant refinement modifier M_{pr} if for each pair $(p, extint)$:

1. p is a participant name in R ;
2. for each operation name m in $extint$: m appears in participant p in R and m 's first specialisation clause in $extint$ is identical to the specialisation clause of m in p in R ;
3. for each operation invocation $a.m$ in a second specialisation clause in $extint$:
   (a) a is an acquaintance name in the acquaintance clause of p in R ;
   (b) m is an operation in the interface of the participant referred to by a in p in R .

Note that the last clause is again a slightly adapted version of well-formedness restrictions *WF2* and *WF3*.

Definition 2.22 (Participant Refinement) If a reuse contract R is participant refinable by a modifier M_{pr} then the reuse contract R_{pr} is the **participant refinement** of R by M_{pr}, where:

1. R_{pr} contains all participants of R that are not mentioned in M_{pr};
2. for each $(p, extint)$ in M_{pr}: R_{pr} contains a participant
   (a) with name p and the same acquaintance clause as p in R ;
   (b) that contains all operations of p in R not mentioned in $extint$;

(c) that contains all operations of *extint* with as specialisation clause the union of their two specialisation clauses in *extint* .

The last clause describes how the participants that are mentioned in M_{pr} are adapted. Their name and acquaintance clause remain the same (clause 2a). The operations in their interfaces that are not in the extended interface attached to p in M_{pr} also remain the same (clause 2b). The operations in their interfaces that are mentioned in the extended interface attached to p in M_{pr} keep their name, but their specialisation clause is extended as specified in extint (clause 2c).

Property 2.5 *A participant refinement of a well-formed reuse contract is well-formed.*

Proof

- WF1 is not influenced by participant refinement;
- WF2 is preserved because the first specialisation clause is identical to the one on the original contract, which was well-formed, while the second specialisation clause respects WF2 because of clause 3a of the refinability definition;
- WF3 is preserved because the first specialisation clause is identical to the one on the original contract, which was well-formed, while the second specialisation clause respects WF3 because of clause 3b of the refinability definition.

Illustration

Figure 2.14 illustrates a participant refinement. Operations on two participants are refined: $name_{11}$ on pcpt1 and $name_{22}$ on pcpt2. To both these operations two specialisation clauses are attached. The first repeats the original specialisation clause, the second states the newly added invocations. Note that on $name_{22}$ the first specialisation clause is empty, because the specialisation clause of $name_{22}$ on the original contract is empty.

Short-Hand Notations

We again introduce short-hand notations describing which operations are refined and which extra operation invocations are added to their specialisation clauses.

Notation 2.11 A participant refinement modifier M_{pr} represents a **participant refinement of** m **on** p if M_{pr} contains a pair $(p, extint)$, with an operation m in *extint* .

Notation 2.12 A participant refinement modifier M_{pr} represents a **participant refinement of** m **on** p **referencing** n **on** q if M_{pr} contains a pair $(p, extint)$, with an operation m in *extint* , with m invoking n on q in m 's second specialisation clause.

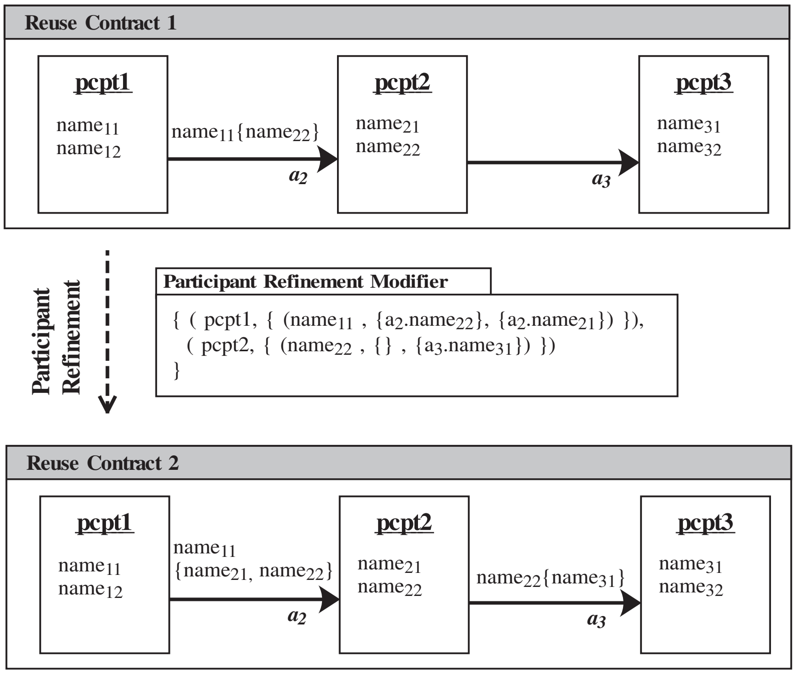

Figure 2.14: Participant Refinement

Note the use of the term "referencing" here. We use this term to denote the addition of both operation invocations and acquaintance relationships by respectively participant and context refinement. The term "dereferencing" is used to denote the removal of operation invocations and acquaintance relationships by respectively participant and context coarsening.

In the case of refinement and coarsening we need some extra notations. The operations in the extended interfaces have two specialisation clauses: one repeating the specialisation clause that is adapted and one stating the operation invocations that are added to the specialisation clause. On the resulting contract, the specialisation clause attached to these operations is the union of both. For refinements, as well as for coarsenings, we need to know the specialisation clause that is repeated, we call this the *repeating specialisation clause* and the specialisation clause that will be attached to this method in the result, we call this the *resulting specialisation clause*. We therefore introduce two extra notations.

Notation 2.13 The **repeating specialisation clause of** an operation m on p in a participant refinement modifier M_{pr} containing a pair $(p, extint)$, with m in $extint$, is the first specialisation clause coupled to m in $extint$.

Notation 2.14 The **resulting specialisation clause of** an operation m on p in a participant refinement modifier M_{pr} containing a pair $(p, extint)$, with m in $extint$, is the union of the two specialisation clauses coupled to m in $extint$.

2.2.6 Context Refinement

Motivation

A contract can be refined by adding operation invocations, but also by adding acquaintance relationships. We call the latter context refinement. Figure 2.15 shows a context refinement, adding a bi-directional binding between `Bank` and `Consortium`. Note that this corresponds to two acquaintance relationships: one on `Consortium` with name `theBank` referring to `Bank` and one on `Bank` with name `theCons` referring to `Consortium`.

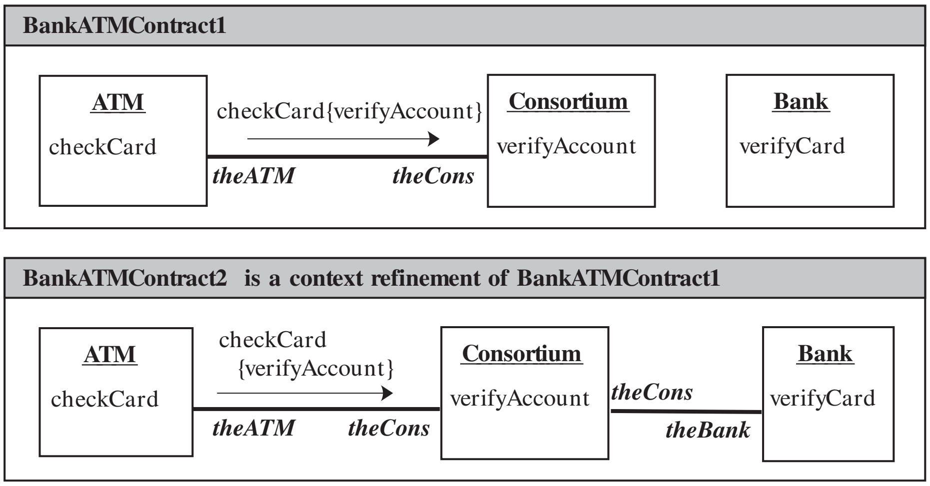

Figure 2.15: An Example Context Refinement

Note that a context refinement can only add acquaintance relationships between existing participants. It cannot add new participants: this needs to be done through context extension.

Definition and Properties

In the same way that we needed to repeat the original specialisation clause in participant refinement modifiers, we need to repeat the original acquaintance clauses here. This explains the form of the modifier.

Definition 2.23 (Context Refinement Modifier) A **context refinement modifier** is a reuse modifier with modifier tag "context refinement" and a modifier description containing triples $(p, acq1, acq2)$ each consisting of a participant name p and two disjoint acquaintance clauses.

The first acquaintance clause repeats the original acquaintance clause, while the second contains the added invocations. It is therefore important for them to be disjoint. Furthermore, all acquaintance names used in the second acquaintance clause have to be different from the ones in the first acquaintance clause, because acquaintance names are unique within acquaintance clause. This is specified in clause 3 below.

Definition 2.24 (Context Refinable) The **context** of a reuse contract R is **refinable** by a context refinement modifier M_{cr} if for each triple $(p, acq1, acq2)$:

1. p is a participant name in R ;
2. acq1 is identical to the acquaintance clause of p in R ;
3. $acq2$ contains acquaintance relationships $a.q$, where a is different from all acquaintance names in $acq1$ and q is a participant name in R .

Definition 2.25 (Context Refinement) If the context of a reuse contract R is refinable by a modifier M_{cr} then the reuse contract R_{cr} is the **context refinement** of R by M_{cr}, where:

1. R_{cr} contains all participants of R that are not mentioned in M_{cr};
2. for each triple $(p, acq1, acq2)$ in M_{cr}: R_{cr} contains a participant with the same name and interface as p in R and the union of $acq1$ and $acq2$ as acquaintance clause.

Context refinement preserves well-formedness.

Property 2.6 *A context refinement of a well-formed reuse contract is well-formed.*

Proof The well-formedness definition imposes 3 constraints.

- WF1 is preserved, because the acquaintance names that already occurred in the acquaintance clause still refer to the same participant and the newly added acquaintance names refer to participants in R because of clause 3 in the refinability definition;
- WF2 and WF3 are respected because nothing is changed to specialisation clauses and no existing acquaintance relations are changed, only new ones are added.

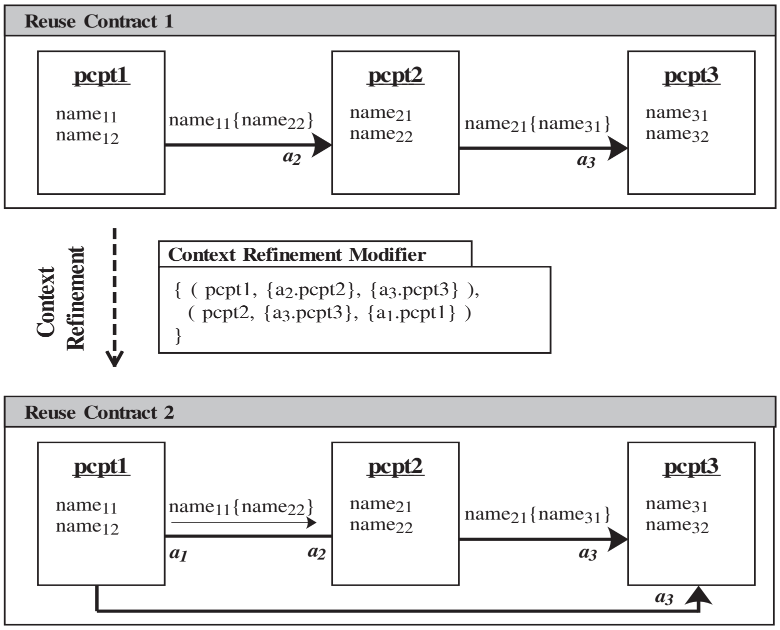

Figure 2.16: Context Refinement

Illustration

Figure 2.16 illustrates a context refinement. Acquaintance relationships are added to two participants: *pcpt*1 and *pcpt*2. There are two acquaintance clauses attached to each participant, because we need to be able to distinguish between acquaintance relationships already present on the original contract and new acquaintance relationships. An acquaintance relationship called a_3 is added from *pcpt*1 to *pcpt*3 and an acquaintance relationship called a_1 from *pcpt*2 to *pcpt*1. Note how this last addition makes the binding between *pcpt*1 and *pcpt*2 bi-directional and causes a change in the graphical notation.

Short-Hand Notations

Notation 2.15 A context refinement modifier M_{cr} represents a **context refinement of** p if M_{cr} contains a triple $(p, acq1, acq2)$.

Notation 2.16 A context refinement modifier M_{cr} represents a **context refinement of** p **referencing** q if M_{cr} contains a triple $(p, acq1, acq2)$ and $acq2$ refers to q .

As we needed short-hands to denote the repeating and resulting specialisation clauses in participant refinement, we need similar short-hands to denote the repeating and resulting aquaintance clauses here.

Notation 2.17 The **repeating aquaintance clause of** a participant p in a context refinement modifier M_{cr} containing $(p, acq1, acq2)$ is $acq1$.

Notation 2.18 The **resulting aquaintance clause of** a participant p in a context refinement modifier M_{cr} containing $(p, acq1, acq2)$ is the union of $acq1$ and $acq2$.

2.2.7 Participant Coarsening

Motivation

The inverse of refinement is called coarsening and entails the removal of dependencies between the model elements. We again start with participant coarsening, which implies the removal of operation invocations. Figure 2.17 shows a participant coarsening: an operation invocation of `processTrAct` by `trActRequest` is removed.

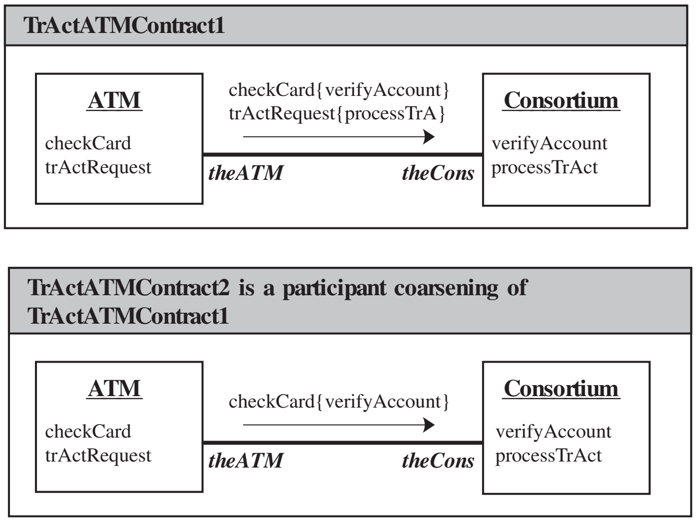

Figure 2.17: An Example Participant Coarsening

Definition and Properties

Again, the form of the coarsening modifier is identical to the form of the refinement modifier. This means that we again work with extended interfaces, where operations have *two* specialisation clauses. The first repeats that part of the specialisation clause that is maintained, while the second indicates which operation invocations need to be removed.

Definition 2.26 (Participant Coarsening Modifier) A **participant coarsening modifier** is a reuse modifier with modifier tag "participant coarsening" and a modifier description containing pairs $(p, extint)$ each consisting of a participant name p and an extended interface $extint$, i.e. a set of operations, each consisting of a name and 2 disjoint specialisation clauses.

Again the disjointness of the two specialisation clauses is important as the first part denotes which invocations are retained and the second which are removed. The applicability of the modifier is stated in the following definition.

Definition 2.27 (Participant Coarsenable) A reuse contract R is **participant coarsenable** by a participant coarsening modifier M_{pc} if for each pair $(p, extint)$:

1. p is a participant name in R ;
2. for each operation name m in $extint$: m appears in participant p in R and the union of m 's specialisation clauses in $extint$ is identical to the specialisation clause of m in p in R ;

Participant coarsening of contracts can then be defined as follows.

Definition 2.28 (Participant Coarsening) If a reuse contract R is participant coarsenable by a modifier M_{pc} then the reuse contract R_{pc} is the **participant coarsening** of R by M_{pc} where:

1. R_{pc} contains all participants of R that are not mentioned in M_{pc};
2. for each pair $(p, extint)$ mentioned in M_{pc}: R_{pc} contains a participant
   (a) with name p and the same acquaintance clause as p in R ;
   (b) that contains all operations of p not mentioned in $extint$;
   (c) that contains all operations of $extint$ with as specialisation clause the first of the two specialisation clauses in $extint$.

As with participant refinement the only change to the participants mentioned in M_{pc} is the specialisation clauses of the operations mentioned in extint.

Property 2.7 *A participant coarsening of a well-formed reuse contract is well-formed.*

Proof

- WF1 is not influenced by participant coarsening;
- WF2 and WF3 are preserved because acquaintance and operation names are only omitted from specialisation clauses, they are not added to them, and the constraint was already respected on the original reuse contract.

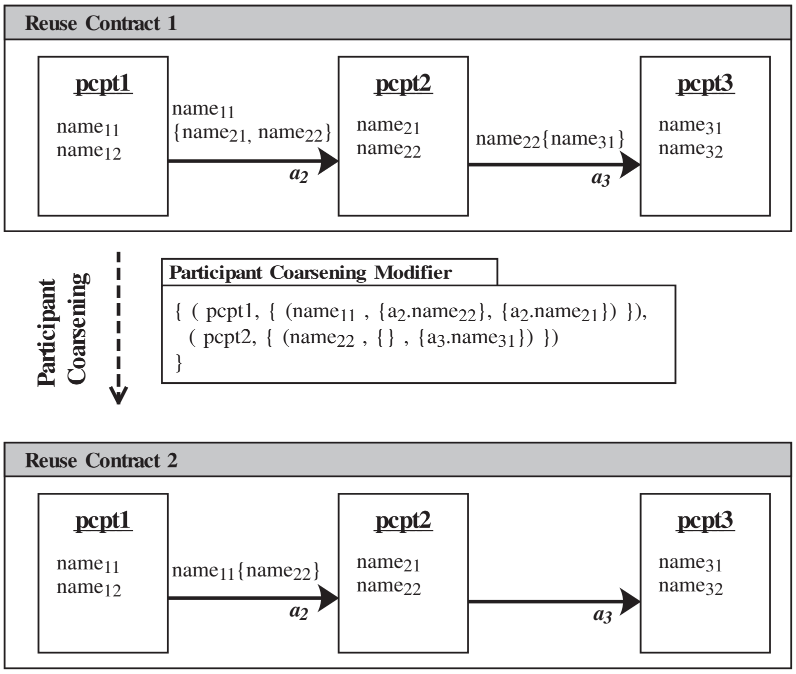

Figure 2.18: Participant Coarsening

Illustration

Figure 2.18 illustrates a participant coarsening. Operations on two participants are removed. All operations in the modifier have two specialisation clauses. Together, they describe the entire specialisation clause on the original contract of the operation they are attached to. The second part is removed on the resulting contract. The first specialisation clause of $name_{22}$ is empty, because the entire specialisation clause of the original contract is removed.

Short-Hand Notations

We again introduce some short-hand notations. Note the use of the word "dereferencing" to denote that a coarsening removes dependencies, as opposed to the use of the word "referencing" that is used with refinements to denote that dependencies are added.

Notation 2.19 A participant coarsening modifier M_{pc} represents a **participant coarsening of** m **on** p if M_{pc} contains a pair (p, int), with an operation m in int .

Notation 2.20 A participant coarsening modifier M_{pc} represents a **participant coarsening of** m **on** p **dereferencing** n **on** q if M_{pc} contains a pair (p, int), with an operation m in int , with m invoking n on q in m 's second specialisation clause.

As with participant refinement we need short-hands to determine the repeating and resulting specialisation clauses of an operation.

Notation 2.21 The **repeating specialisation clause of** an operation m on p in a participant coarsening modifier M_{pc} containing a pair $(p, extint)$, with m in $extint$, is the union of the two specialisation clauses coupled to m in $extint$.

Notation 2.22 The **resulting specialisation clause of** an operation m on p in a participant coarsening modifier M_{pc} containing a pair $(p, extint)$, with m in $extint$, is the first specialisation clause coupled to m in $extint$.

2.2.8 Context Coarsening

Motivation

Again, a contract cannot only be coarsened by removing operation invocations, but also by removing acquaintance relationships. We call this lat operator context coarsening. Figure 2.19 shows a context coarsening, the two acquaintance relationships that together form the bi-directional binding between `Bank` and `Consortium` are removed.

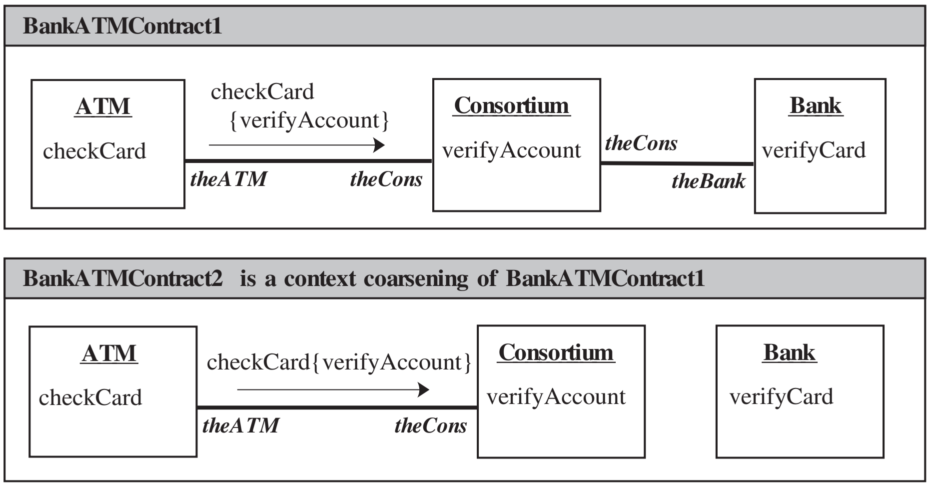

Figure 2.19: An Example Context Coarsening

Note that this acquaintance relationship can only be removed because there are no operation invocations modelled along it. The acquaintance relationship between `ATM` and `Consortium` can, for example, not be removed. (At least not in that direction. Note that there is a bi-directional binding between `ATM` and `Consortium` and we can make it uni-directional by removing the acquaintance relationship *from* `Consortium` *to* `ATM`.)

Definition and Properties

A context coarsening modifier has the same form as a context refinement modifier, meaning that it contains two acquaintance clauses, for a number of participant names. The first part contains the acquaintance relationships to be maintained, the second part the acquaintance relationships to be removed.

Definition 2.29 (Context Coarsening Modifier) A **context coarsening modifier** is a reuse modifier with modifier tag "context coarsening" and a modifier description containing triples $(p, acq1, acq2)$ each consisting of a participant name p and two disjoint acquaintance clauses.

Again, it is important that the two acquaintance clauses are disjoint, because the first part will be maintained and the second part removed.

Definition 2.30 (Context Coarsenable) The **context** of a reuse contract R is **coarsenable** by context coarsening modifier M_{cc} if for each triple $(p, acq1, acq2)$:

1. p is a participant name in R ;
2. the union of $acq1$ and $acq2$ is identical to the acquaintance clause of p in R ;
3. for all $a.q$ in $acq2$: no operation in p has a in its specialisation clause.

We can now define context coarsening as follows:

Definition 2.31 (Context Coarsening) If a reuse contract the context of R is coarsenable by a modifier M_{cc} then the reuse contract R_{cc} is the **context coarsening** of R by M_{cc}, where:

1. R_{cc} contains all participants of R that are not mentioned in M_{cc};
2. for each triple $(p, acq1, acq2)$ in M_{cc}: R_{cc} contains a participant with the same name and interface as p in R and $acq1$ as acquaintance clause.

Property 2.8 *A context coarsening of a well-formed reuse contract is well-formed.*

Proof

- WF1 is respected because no new acquaintance relationships are added and no participants are removed;
- WF2 is respected through the fact that only acquaintance relationships can be removed, along which no operation invocations are modelled. This is enforced by the restriction in the third clause of the context coarsenability definition;

- because of the integrity of WF2, WF3 is not affected by this operator.

Illustration

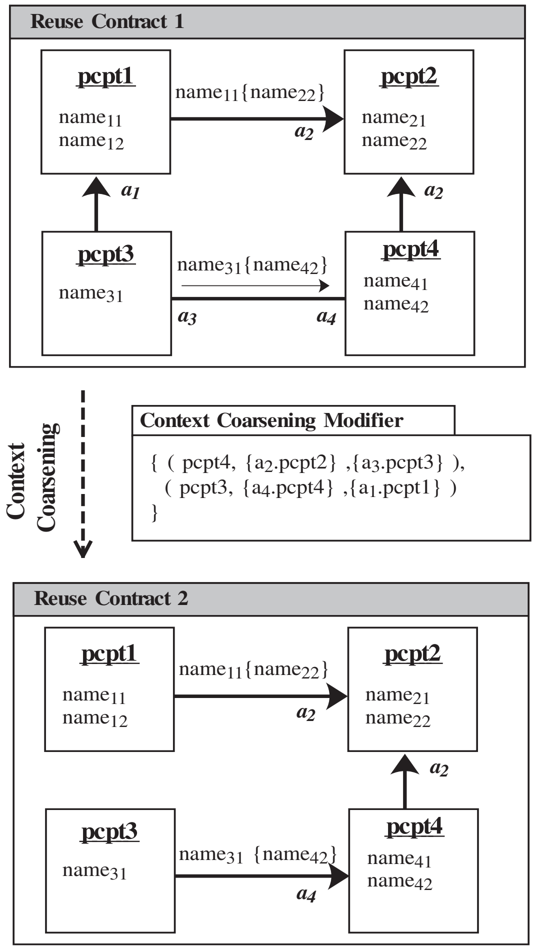

Figure 2.20: Context Coarsening

Figure 2.20 illustrates a context coarsening. The acquaintance clauses of two participants are coarsened. From pcpt4 the acquaintance relationship with pcpt3 is removed, from pcpt3 the relationship with pcpt1. Note that the union of the two acquaintance clauses attached to each participant name in the modifier forms the complete acquaintance clause of the participant on the original contract. Also note that in the original contract no operation invocations were modelled along the acquaintance relationships in the second parts. The operation $name_{42}$ is invoked by

$name_{31}$ along the acquaintance relationship a_4 of pcpt3, not along the acquaintance a_3 of pcpt4.

Short-Hand Notations

Notation 2.23 A context coarsening modifier M_{cc} represents a **context coarsening of** p if M_{cc} contains a triple $(p, acq1, acq2)$.

Notation 2.24 A context coarsening modifier M_{cc} represents a **context coarsening of** p **dereferencing** q if M_{cc} contains a triple $(p, acq1, acq2)$ and $acq2$ refers to q .

We again introduce short-hands for repeating and resulting aquaintance clauses.

Notation 2.25 The **repeating aquaintance clause of** a participant p in a context coarsening modifier M_{cc} containing $(p, acq1, acq2)$ is the union of $acq1$ and $acq2$.

Notation 2.26 The **resulting aquaintance clause of** a participant p in a context coarsening modifier M_{cc} containing $(p, acq1, acq2)$ is $acq1$.

2.2.9 Summary

We introduced eight orthogonal and elementary operators on reuse contracts that together are complete in that they can model all possible adaptations to a reuse contract. Participant and context extension respectively add operations and participants to a contract, while participant and context cancellation remove them. Participant and context refinement respectively add operation invocations and acquaintance relationships, while participant and context coarsening remove them. This was depicted in table 2.1 on page 49.

All operators were defined by means of a modifier definition, an applicability definition and an operator definition. The first describes what a modifier for this particular operator looks like, independent of the reuse contract it will be applied to. The second defines what constraints a modifier must fulfil in order to be applicable to a particular reuse contract. The third defines how the result of applying this operator to a well-formed reuse contract can be computed. We only apply operators to well-formed reuse contracts and all operators share the property that they preserve well-formedness.

A number of the basic reuse modifiers repeat information about the reuse contract they are applied to. We said that was necessary in order to detect certain conflicts. In the next chapter we discuss how the operators can assist in detecting conflicts upon evolution and composition. We also discuss the trade-off between adding more information to reuse modifiers, thus being able to detect more conflicts, and adding less information, thus providing more flexibility.