Vrije Universiteit Brussel
Faculteit Wetenschappen

Formalising Operations on ACIDs and Their Interactions

Kim Mens, Carine Lucas, Patrick Steyaert

Techreport vub-prog-tr-96-03

Programming Technology Lab
PROG(WE)
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3525
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

Formalising Operations on ACIDs and Their Interactions -- DRAFT[1] - Do not distribute --

Kim Mens, Carine Lucas, Patrick Steyaert

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
http://progwww.vub.ac.be/

Email: kimmens@is1.vub.ac.be, clucas@vnet3.vub.ac.be, prsteyae@vnet3.vub.ac.be

Abstract *This paper provides a formal foundation of the concept of Abstract Class Interface Descriptions (ACIDs). It gives a definition of both ACIDs and the operations defined on them and proves a number of properties concerning their interactions. In class libraries and frameworks documented with ACIDs these properties can be used to provide a better understanding of their layered structure and to help in assessing the impact of changes.*

1. Introduction

This technical report formally introduces ACIDs, operations on ACIDs and their interactions on a formal level. For a more intuitive discussion of ACIDs and their interactions and of how they can be used in object-oriented software engineering in general, we refer the reader to [Steyaert&al.96].

2. ACIDs and Operations on ACIDs

2.1 Definition of ACIDs

Every ACID A is an interface, i.e. a set of method signatures. To every method signature a (possibly empty) specialisation clause is attached. Furthermore, to every method signature an annotation ‘abstract’ or ‘concrete’ is attached.
We define the following selector functions on an ACID A:

Client(A)	=	set of all method signatures in A (without the attached specialisation clauses and without the attached annotation ‘abstract’ or ‘concrete’)
Abstract(A)	=	set of all abstract method signatures of A
Concrete(A)	=	set of all concrete method signatures of A

[1] This technical report is currently being finalised. This version provides proofs for the main properties discussed in [Steyaert&al96], but lacks binding text and some additional proofs. A final version will be made public in the near future.

$\mathrm{Spec}_A(m)$	=	specialisation clause corresponding to the method signature m in A
$\mathrm{Annot}_A(m)$	=	annotation corresponding to the method signature m in A

Following property follows directly from the definition:

Property:
For every ACID A: Abstract(A) $\cap$ Concrete(A) = $\emptyset$

Furthermore, an ACID is well-formed if every method appearing in one of its specialisation clauses also appears in the client interface of the ACID:

An ACID A is **well-formed** if:
$\forall$ m $\in$ Client(A): $\forall$ n $\in$ Spec_A(m): n $\in$ Client(A)

2.2 Definition of applicability of operations on ACIDs

In the following section we will define a number of operations on ACIDs in terms of an increment M. For such an M to be a correct increment to perform a certain operation on an ACID, it needs to comply to certain properties. We speak of the *applicability* of M. Furthermore, depending on the operation, the increments M contain different information (names, specialisation clauses, an annotation abstract or concrete). Therefore, the prerequisites for each operation do not only describe the conditions that the increment must comply to in order to be correct, but also the kind of information given by this increment. There are thr[redacted], corresponding to the three basic operations on ACIDs. In the foll ing definitions, consider A, A_X to be ACIDs:

A **is concretisabl with** M $\Leftrightarrow$
(1) M is a set of meth
(2) Client(M) __ Abstract(A)

If this subset-relationship is strict, we say that M yields a partial c[redacted]retisation; if the two sets are equal we say that M yields a complete concretisation. More rmally,

M yields a **complete** concretisation of A	$\Leftrightarrow$	Client(M) = Abstr
M yields a **partial** concretisation of A	$\Leftrightarrow$	Client(M) Abstract(A)

In the above definition and the following proofs we often need to refer to the set of all signatures of methods in M. In order to obtain these signatures, we will write Client(M), as we do for ACIDs. In the above case of a concretising M, ‘Client’ is simply defined as the identity function, since M consists of method signatures only.

A **is extendible with** M ⇔
(1) M is an ACID, i.e. a set of meth[illegible] signatures with a specialisation clause and an annotation abstract or concret[illegible]tached to each one of them
(2) Client(M) $\cap$ Client(A) = $\emptyset$
(3) $\forall$ m $\in$ Client(M) : Spec$_M$(m) __ Client(M) $\cup$ Client(A)

If M contains only concrete methods, we say that M yields a concrete extension, otherwise we say that M yields an abstract extension. I.e.

M yields a **concrete** extension of A	⇔	Abstract(M) = $\emptyset$
M yields an **abstract** extension of A	⇔	Abstract(M) $\neq \emptyset$

Again, the notation Client(M) is used for an extending M to retrieve the set of all method signatures of M. We also overload the notation Spec$_M$(m) to denote the specialisation clause associated to a signature m in M.

A **is refinable with** [illegible] ⇔
(1) M is compos[illegible]t of two parts: M$_e$ wh[illegible]nd M$_r$ which contains only method [illegible]natures with accorde[illegible]uses (but no annotation abstract or co[illegible]ete).
(2) Client(M$_e$) $\cap$ Clie[illegible]
(3) Client(M$_r$) __ Client(A)
(4) $\forall$ m $\in$ Client(M$_r$) : Spec*$_A$(m) [illegible]
(5) $\forall$ m $\in$ Client(M) : Spec$_M$(m) __ Client(M) $\cup$ Client(A)
(6) $\forall$ m $\in$ Client(M$_e$) : $\exists$ n $\in$ Client(M$_r$) : m $\in$ Spec*$_{A,M}$(n)

The two parts of the increment M represent the two parts of the refinement's functionality. M$_r$ indicates which methods have their specialisation clauses refined and how, while M$_e$ indicates which methods are added.
If the M$_e$ part of M contains only concrete methods, we say that M yields a concrete refinement, otherwise we say that M yields an abstract refinement.

M = (M$_e$,M$_r$) yields a **concrete** refinement of A	⇔	Abstract(M$_e$) = $\emptyset$
M = (M$_e$,M$_r$) yields an **abstract** refinement of A	⇔	Abstract(M$_e$) $\neq \emptyset$

Several remarks need to be made about the notations used in the above definition. First the set of all method signatures of M is the union of all method signatures in M$_e$ and M$_r$: Client(M) = Client(M$_e$) $\cup$ Client(M$_r$), where Client(M$_e$) and Client(M$_r$) merely select the set of all method signatures of M$_e$ and M$_r$ respectively. The same remark can be made about the selector functions Spec and Annot. Secondly, in analogy with selecting the specialisation clause Spec$_A$(m) of a method signature m in an ACID A, Spec$_{A,M}$(m) is used to find the specialisation clause of m in the ACID which results from refining A with M, and is defined as follows:

$$\begin{array}{llll}
Spec_{A,M}(m) & = Spec_{Me}(m) & \text{if} & m \in Client(M_e) \\
 & = Spec_{Mr}(m) & \text{if} & m \in Client(M_r) \\
 & = Spec_{A}(m) & \text{if} & m \in Client(A) - Client(M)
\end{array}$$

Finally, the operator * used above denotes transitive closure and is defined as usual:

$$\begin{array}{lll}
f^*(m) = \cup_{n\geq 1} f^n(m) & \text{where} & f^1(m) = f(m) \\
 & & f^n(m) = \{ m' \in f(n) \mid n \in f^{n-1}(m) \}
\end{array}$$

2.3 Definition of operations on ACIDs

This section now formally introduces the three basic operations on ACIDs of which the applicability rules were defined in the previous section. Of course, the first prerequisite is always that the increment M has to be applicable. The other ones indicate how the resulting ACID is to be constructed. As in the previous definitions, we consider all A and A_x to be ACIDs:

A_c **is a concretisation of** A **with** M $\Leftrightarrow$

(1) A is concretisable with M

(2) $Client(A_c) = Client(A)$

(3) $\forall\, m \in Client(A_c) : Spec_{Ac}(m) = Spec_A(m)$

(4) $\forall\, m \in Client(M) : Annot_{Ac}(m) = \text{‘concrete’}$

(5) $\forall\, m \in Client(A_c) - Client(M) : Annot_{Ac}(m) = Annot_A(m)$

The following property follows immediately from the definition:

Lemma 1: If A_c is a concretisation of A with M then

(a) $Concrete(A_c) = Concrete(A) \cup Client(M)$

(b) $Abstract(A_c) = Abstract(A) - Client(M)$

A_e **is an extension of** A **with** M $\Leftrightarrow$

(1) A is extendible with M

(2) $Client(A_e) = Client(A) \cup Client(M)$

(3) $\forall\, m \in Client(A) : Spec_{Ae}(m) = Spec_A(m) \wedge Annot_{Ae}(m) = Annot_A(m)$

(4) $\forall\, m \in Client(M) : Spec_{Ae}(m) = Spec_M(m) \wedge Annot_{Ae}(m) = Annot_M(m)$

A_r **is a refinement of** A **with** M $= (M_e, M_r) \Leftrightarrow$

(1) A is refinable with M

(2) $Client(A_r) = Client(A) \cup Client(M_e)$

(3) $\forall\, m \in Client(M) : Spec_{Ar}(m) = Spec_M(m)$

(4) $\forall\, m \in Client(A_r) - Client(M) : Spec_{Ar}(m) = Spec_A(m)$

(5) $\forall\, m \in Client(M_e) : Annot_{Ar}(m) = Annot_{Me}(m)$

(6) $\forall\, m \in Client(A_r) - Client(M_e) : Annot_{Ar}(m) = Annot_A(m)$

The following property follows immediately from this definition:

Lemma 2: If A_r is a refinement of A with M = (M_e,M_r) then
Abstract(A_r) = Abstract(A) ∪ Abstract(M_e)
= Abstract(A) ∪ (Abstract(M) - Client(A))

2.4 Definition of applicability of inverse operations on ACIDs

For each of the three operations on ACIDs, the inverse operation is defined. Again we start by defining applicability rules for these inverse operations and we consider A, A_x to be ACIDs in the following definitions. Furthermore, as in section 2.2 we will overload the relations ‘Client’ and ‘Spec’ for selecting the client interface, resp. specialisation clauses of the inverse “in[redacted]ements”.

Abstraction is the inve[redacted] of concretisation.

A **is abstractable** [redacted]**ith** M ⇔
(1) M is a set of meth[redacted]
(2) Client(M) __ C[redacted]

Cancellation is the i[redacted]erse operation[redacted]f exte[redacted]ion.

A **is cancellab**[redacted] **with** M [redacted]
(1) M is a set of meth[redacted]
(2) Client(M) __ Client(A)
(3) ∀ m ∈ Client(M) : [redacted]lient(M) : m ∈ Spec_A(n)

Coarsening is the inverse operation [redacted] refinement.

A **is coarsenable** [redacted]**th** M = (M_{ca},M_{co}) ⇔ [redacted]
(1) M is compos[redacted] out of two parts: M[redacted]ich contains only method signatures and M_{co} whi[redacted] contains method sig[redacted]tures with their accorded specialisation clauses (but no an[redacted]ncrete).
(2) Client(M) __ Client(A)
(3) ∀ m ∈ Client(M_{co}) : Spec_{Mco}(m) Spec_A(m)
(4) ∀ m ∈ Client(M_{ca}) : ∀ n ∈ Client(A) with m ∈ Spec_A(n):
n ∈ Client(M_{co}) ∧ m ∉ Spec_{Mco}(n)

2.5 Definition of inverse operations on ACIDs

For each of the three operations on ACIDs, the inverse operation is defined. As for the base operations the definitions of the inverse operations start with an applicability constraint, followed by a number of predicates explaining how the resulting ACID is constructed.

A_c^- **is an abstraction of** A **with** M $\Leftrightarrow$

(1) A is abstractable with M

(2) $Client(A_c^-) = Client(A)$

(3) $\forall\, m \in Client(A) : Spec_{A_c^-}(m) = Spec_A(m)$

(4) $\forall\, m \in Client(M) : Annot_{A_c^-}(m) =$ ‘abstract’

(5) $\forall\, m \in Client(A_c) - Client(M) : Annot_{A_c}(m) = Annot_A(m)$

The following property follows immediately from this definition:

Lemma 3:

(a) $Abstract(A_c^-) = Abstract(A) \cup Client(M)$

(b) $Concrete(A_c^-) = Concrete(A) - Client(M)$

A_e^- **is a cancellation of** A **with** M $\Leftrightarrow$

(1) A is cancellable with M

(2) $Client(A_e^-) = Client(A) - Client(M)$

(3) $\forall\, m \in Client(A_e^-) : Spec_{A_e^-}(m) = Spec_A(m) \wedge Annot_{A_e^-}(m) = Annot_A(m)$

Coarsening is the inverse of refinement.

A_r^- **is a coarsening of** A **with** $M = (M_{ca}, M_{co})$ $\Leftrightarrow$

(1) A is coarsenable with M

(2) $Client(A_r^-) = Client(A) – Client(M_{ca})$

(3) $\forall\, m \in Client(M_{co}) : Spec_{A_r^-}(m) = Spec_{M_{co}}(m)$

(4) $\forall\, m \in Client(A_r^-) - Client(M_{co}) : Spec_{A_r^-}(m) = Spec_A(m)$

(5) $\forall\, m \in Client(A_r^-) : Annot_{A_r^-}(m) = Annot_A(m)$

3. Basic Interactions between Operations on ACIDs

We will now prove a number of properties concerning the interactions between these operations. In this section we discuss the properties concerning base ACID exchange as discussed in [Steyaert&al.96].

3.1 Applicability

The first range of questions we need to answer concerns the applicability of the operations. We want to investigate whether an increment M, that was applied on a base ACID to create an application ACID, is still applicable to an exchanged base ACID. We therefore use the three applicability definitions (is concretisable with, is extendible with, is refinable with) that were given in section 2.3. The fact that an increment is no longer applicable after base ACID exchange can only be due to name clashes in the client interface. The following table summarises under which conditions such name clashes occur.

Operation on base ACID / *Operation on application ACID*	**Concretisation**	**Extension**	**Refinement**
Concretisation	if sets of concretised method signatures intersect	no clashes	no clashes
Extension	no clashes	if sets of newly added method signatures intersect	if sets of newly added method signatures intersect
Refinement	no clashes	if sets of newly added method signatures intersect	if sets of newly added method signatures intersect

It demonstrates that essentially 3 categories of interactions can be distinguished. We will discuss these 3 cases one by one.

3.1.1 Concretisation versus Concretisation

Property 1:

If A is concretisable with M_1
and A_2 is a concretisation of A with M_2
then (A_2 is concretisable with M_1 ⇔ Client(M_1) ∩ Client(M_2) = ∅)

Proof:

Suppose that A is concret[illegible]

A_2 is concretisable wit[illegible] M_1
⇔ M_1 is a se[illegible]f meth[illegible] (o.k. because A is conc[illegible]isable with M_1)
Client(M_1) __ A[illegible]
⇔ Client(M_1) __ Abstract(A) - Client(M_2) (lemma 1)
⇔ Client(M_1) ∩ Client(M_2) = ∅ (because Client(M_2) __ Abstract(A), since A is concretisable with M_1)

3.1.2 Concretisation versus Refinement / Extension

Property 2a:

If A is concretisable with M_1 and A is extendible or refinable with M_2
then (A_1 is a concretisation of A with M_1
⇒ A_1 is extendible or refinable with M_2)

This property also holds in the reversed direction.

Property 2b:

If A is concretisable with M_1 and A is extendible or refinable with M_2
then (A_2 is an extension or refinement of A with M_2
⇒ A_2 is concretisable with M_1)

We give the proof for one sub-case of the first property and leave the other (analogous) proofs to the reader.

Proof:

We know that:

A_1 is a concretisation of A with $M_1 \Rightarrow \text{Client}(A_1) = \text{Client}(A)$

Furthermore, we know that

A is extendible with M_2 $\Leftrightarrow$

(1) M_2 is an ACID

(2) $\text{Client}(M_2) \cap \text{Client}(A) = \emptyset$

(3) $\forall\, m \in \text{Client}(M_2) : \text{Spec}_{M_2}(m)$ __ $\text{Client}(M_2) \cup \text{Client}(A)$

For the property to hold we need to prove that A_1 is extendible with M_2, in other words we need to prove that:

(1) M_2 is an ACID — extendible with M_2

(2) $\text{Client}(M_2) \cap \text{Client}(A_1) = \emptyset$

$\Leftrightarrow \text{Client}(M_2) \cap \text{Client}(A) = \emptyset$ — as $\text{Client}(A_1) = \text{Client}(A)$

A is extendible with M_2

(3) $\forall\, m \in \text{Client}(M_2)$: $\text{Spec}_{M_2}(m)$ __ $\text{Client}(M_2) \cup \text{Client}(A_1)$

$\Leftrightarrow \forall\, m \in \text{Client}(M_2)$: $\text{Spec}_{M_2}(m)$ __ $\text{Client}(M_2) \cup \text{Client}(A)$

as $\text{Client}(A_1) = \text{Client}(A)$

this holds as A is extendible with M_2

3.1.3 Refinement / Extension versus Refinement / Extension

Property 3:

If A is extendible or refinable with M_1

and A_2 is an extension or refinement of A with M_2

then (A_2 is extendible or refinable with M_1 $\Leftrightarrow$ $\text{Client}(M_1) \cap \text{Client}(M_2) = \emptyset$)

We give the proof for two extensions, the other (analogous) proofs are left to the reader.

Proof:

We know that:

A_2 is an extension of A with $M_2 \Rightarrow \text{Client}(A_2) = \text{Client}(A) \cup \text{Client}(M_2)$

Furthermore, we know that

A is extendible with M_1 $\Leftrightarrow$

(1) M_1 is an ACID

(2) $\text{Client}(M_1) \cap \text{Client}(A) = \emptyset$

(3) $\forall\, m \in \text{Client}(M_1) : \text{Spec}_{M_1}(m)$ __ $\text{Client}(M_1) \cup \text{Client}(A)$

To prove the property we need to show:

A_2 is extendible or refinable with M_1 $\Leftrightarrow$ $\text{Client}(M_1) \cap \text{Client}(M_2) = \emptyset$

A_2 is extendible with M_1 ⇔

(1) M_1 is an ACID [redacted]s as A is extendible with M_1

(2) $Client(M_1) \cap Client(A_2) = \varnothing$ [redacted]

⇔ $Client(M_1) \cap (Client(A) \cup$ Cli[redacted]t(M[redacted]) = ∅   as $Client(A_2)=Client(A) \cup Client(M_2)$

⇔ $Client(M_1) \cap Client(M_2) = \varnothing$ [redacted] this holds because Client([redacted] = ∅

a[redacted] is extendible with M_1

(3) ∀ m ∈ $Client(M_1)$: $Spec_{M1}(m)$ __ Client([redacted]

⇔ ∀ m ∈ $Client(M_1)$: $Spec_{M1}(m)$ __ $Client(M_1) \cup$ (Client(A[redacted]

this holds because ∀ m ∈ $Client(M_1)$: $Spec^*_{M1}(m)$ __ $(Client(M_1) \cup Client(A))$

as A is extendible with M_1

So the only condition left is indeed

$Client(M_1) \cap Client(M_2) = \varnothing$

3.2 Partial concretisations

The second problem is that of invoking unimplemented methods. This occurs when a base ACID is exchanged with a refined or extended version that adds new abstract method signatures. In general, we can say:

Property 4:

If A_c is a concretisation of A with M_c
and A_r is a refinement or an extension of A with M_r
then A_{rc} is a concretisation of A_r with M_c.

Furthermore:

Property 5:

Only if A_c is a *complete* concretisation of A with M_c
and A_r is a *concrete* refinement or extension of A with M_r
then A_{rc} is a *complete* concretisation of A_r with M_c.

Proof:

This proof contains two parts.

First, the fact that M_C still provides a correct concretisation of A_r. This was already demonstrated by property 2b.

Second, we need to proof that if M_C provides a *complete* concretisation of A and M_r a *concrete* refinement or extension of A, than M_C also provides a *complete* concretisation of A_r.

Given:

M_C provides a *complete* concretisation A, in other words $Client(M_c) = Abstract(A)$

M_r a *concrete* refinement or extension of A, in other words $Abstract(M_r) = \varnothing$

To proof:

M_c provides a *complete* concretisation A, in other words $Client(M_c) = Abstract(A_r)$

Proof:

$Abstract(A_r)$	$=$	$Abstract(A) \cup (Abstract(M_r) - Client(A))$	(lemma)
	$=$	$Abstract(A) \cup (\emptyset - Client(A))$	(M_r is concrete ref. or ext.)
	$=$	$Abstract(A)$	
	$=$	$Client(M_c)$	(M_c is complete concretisation of A)

3.3 Detection of Method Capture

Method capture occurs on base ACID exchange, if the exchanged ACID names a certain method m in its specialisation clauses more often than the original base ACID did. To describe the detection of method capture we first need to introduce a new definition.

We say that a method m is bound by a method n in an ACID A, if m appears in the specialisation clause of n in A. We define $MB_A(m)$ as the set of all methods that "bind" m in A.

Definition: $MB_A(m) = \{n \in Client(A) \mid m \in Spec_A(n) \}$

Method capture occurs when extra bindings of a method m are introduced when going from one base ACID A_1 to another base ACID A_2 and this method m was already adapted in some way by an application ACID which applied the increment M_{app} to A_1. In other words, a method m is captured if when changing ACID A_1 to ACID A_2, extra methods are added and m is a member of M_{app}. $MC(A_1,A_2,M_{app})$ denotes the set of all signatures of such methods.

Definition:

$MC(A_1,A_2,M_{app}) = \{ m \in Client(M_{app}) \mid MB_{A_2}(m) - MB_{A_1}(m) \neq \emptyset \}$

As method capture can only occur when specialisation clauses are extended, it can only happen when exchanging the base ACID with a refinement or extension. More specifically, method capture only occurs when the set of hook methods that was added through refinement or extension to the base ACID and the set of method signatures added or changed by the application ACID are not disjoint.

Figure 1 illustrates the detection of method capture.

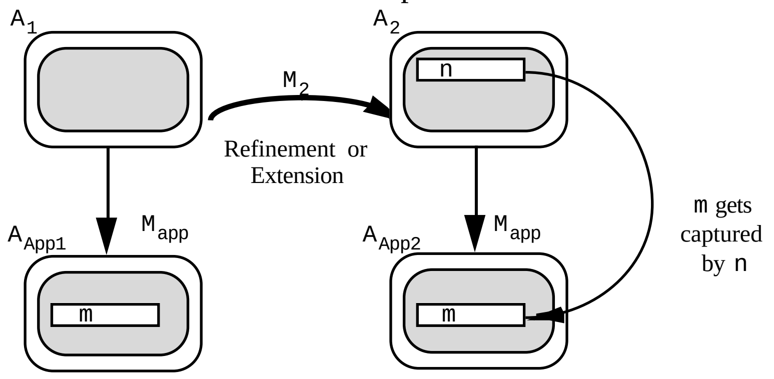

Figure 1: Method Capture

We will now proof that method capture can only occur when a base ACID is exchanged with a refinement or an extension. This done by proving the following

A_2 is obtained from A_1 by an operation different from extension or refinement $\Rightarrow$ $MC(A_1,A_2,M_{app}) = \emptyset$

Proof:

There are 6 possible operations, that can turn A_1 into A_2. We will demonstrate that method capture cannot occur with the operations different from extension and refinement. For these operations we will show that $MC(A_1,A_2,M_{app}) = \emptyset$ or more specifically that

$\forall\ m \in \text{Client}(M_{app}): MB_{A_2}(m) - MB_{A_1}(m) = \emptyset$

To do this it is sufficient to consider all $m \in A_1$ in the following proof, instead of all $m \in \text{Client}(M_{app})$.

First, because for all operations except extension and refinement Client(A_2) __ Client(A_1), or in other words A_2 does not introduce any new methods.

Second, because a method $m \in M_{app}$ cannot be captured by A_2 unless this method already appears in A_2 itself. The reason for this is that for an ACID to be well-formed it can only name methods in its specialisation clauses that appear in the ACID's client interface as well.

We will now show for the 4 remaining operations that:

$\forall\ m \in \text{Client}(A_1): MB_{A_2}(m) - MB_{A_1}(m) = \emptyset$

(i) A_2 is a concretisation of A_1:

$\Rightarrow$ $\text{Client}(A_1) = \text{Client}(A_2)$
and $\forall\ m \in \text{Client}(A_2): Spec_{A_2}(m) = Spec_{A_1}(m)$

$\Rightarrow$ $\forall\ m \in \text{Client}(A_2): MB_{A_2}(m) = MB_{A_1}(m)$

$\Rightarrow$ $\forall\ m \in \text{Client}(A_1) : MB_{A_2}(m) - MB_{A_1}(m) = \emptyset$ // as $\text{Client}(A_2) = \text{Client}(A_1)$

(ii) A_2 is an abstraction of A_1:

Exactly the same reasoning as for concretisati[illegible] holds here.

(iii) A_2 is a cancellation of A_1 with M_2:

$\text{Client}(A_2) = \text{Client}(A_1) - \text{Client}(M_2)$

$\Rightarrow$ $\forall\ m \in \text{Client}(A_1) - \text{Client}(M_2)$: Sp[illegible]ec$_{A_1}(m)$

$\Rightarrow \forall\ m \in \text{Client}(A_1) - \text{Client}(M_2)$[illegible]MB$_{A_2}$[illegible]n) __ $MB_{A_1}(m)$

$\forall\ m \in \text{Client}(M_2) : m \notin \text{Client}(A_2)$

$\Rightarrow \forall\ m \in \text{Client}(M_2)$: [illegible]n $\in Spec_{A_2}(n)$

$\Rightarrow \forall\ m \in \text{Client}(M_2) : MB_{A_2}(m)$[illegible]$\emptyset$

$\Rightarrow$ $\forall\ m \in \text{Client}(A_1) : MB_{A_2}(m) - MB_{A_1}(m) = \emptyset$

(iv) A_2 is a coarsening of A_1 with $M = (M_{ca},$ [redacted]*):*

$\Rightarrow$ $\forall m \in Client(M_{co}) : Spec_{A_2}(m) =$ [redacted]

and $Spec_{M_{co}}(m)$ __ [redacted] nition coarsenable

and $Client(A_2)$ __ $Client(A_1)$ [redacted] coarsening

$\Rightarrow$ $\forall m \in Client(M_{co}) : MB_{A_2}(m)$ __ $MB_{A_1}(m)$

$\forall m \in Client(A_2) - Client(M_{co}): Spec_{A_2}(m) = Spec_{A_1}(m):$ [redacted]

$\Rightarrow$ $\forall m \in Client(A_2) - Client(M_{co}) : MB_{A_2}(m)$ __ $MB_{A_1}(m)$

$\forall m \in Client(M_{ca}) : m \notin Client(A_2)$

$\Rightarrow \forall m \in Client(M_{ca}) : MB_{A_2}(m) = \emptyset$

$\Rightarrow \forall m \in Client(A_1) : MB_{A_2}(m) - MB_{A_1}(m) = \emptyset$

Thus method capture can only occur when a base ACID is changed through a refinement or an extension.

3.4 Detection of Inconsistent Methods

Inconsistent methods occur when there are less bindings of a method m after going from one ACID to another ACID and this method m was adapted in some way when creating an application ACID with the increment M_{app}. In other words, a method m becomes inconsistent if when changing ACID A_1 to ACID A_2, there exists a method that is no longer an element of $MB_{A_2}(m)$ and m is a member of M_{app} (this is denoted: $m \in IM(A_1,A_2,M_{app})$).

Definition:
$IM(A_1,A_2,M_{app}) = \{ m \in M_{app} \mid MB_{A_1}(m) - MB_{A_2}(m) \neq \emptyset\}$

The set of methods that m becomes inconsistent with can be denoted:

Definition:
$IMSet\ (A_1,A_2,m) = MB_{A_1}(m) - MB_{A_2}(m)$

While method capture can occur when extending the specialisation clauses in a base ACID, inconsistent methods can be created when parts of the design are omitted by narrowing these specialisation clauses. This can only be achieved through the operations coarsening and cancellation. Cancellation however does not create inconsistencies, as the method signature that omitted the reference from its specialisation clause simply does not exist anymore. Inconsistent methods can thus only appear when the set of hook methods removed from the base ACID through coarsening and the set of method signatures changed or added by the application ACID are not disjoint.
Figure 2 illustrates the above rule on detection of inconsistent methods.

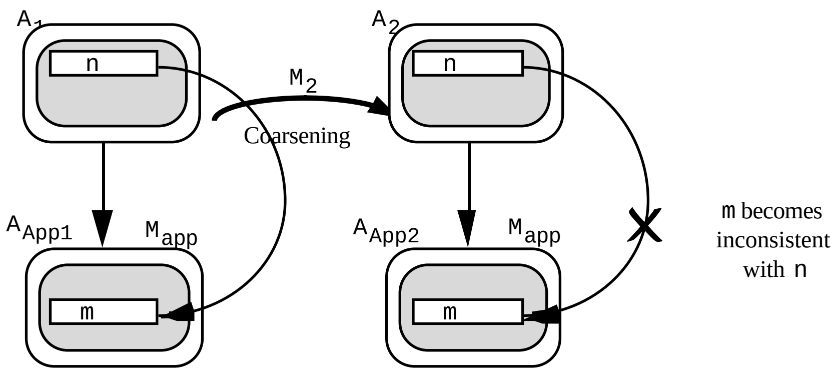

Figure 2: Inconsistent Methods

We will proof that inconsistent methods can only occur through coarsening, by proving the following:

A_2 is obtained from A_1 by an operation different from coarsening $\Rightarrow$
IM $(A_1,A_2,M_{app}) = \emptyset$

Proof:

Again there are 6 possible operations, that can turn A_1 in to A_2. It can be demonstrated in the same way as for method capture that inconsistent methods can only occur through coarsening.

Cancellation is a special case. As coarsening, it also removes names from specialisation clauses, but as all the names that are removed from specialisation clauses through cancellation are removed from the client interface of the ACID as well no inconsistencies can be created.

The proofs for other operations are straightforward.

5. References

[Steyaert&al.96]	Patrick Steyaert, Carine Lucas, Kim Mens, Theo D'Hondt: *Abstract Class Interface Descriptions (ACIDs): Guiding Design Reuse in Class Libraries*, Submitted to OOPSLA '96, Conference on Object-Oriented Programming, Systems, Languages and Applications.