Capsules and types in Fresco

Program verification in Smalltalk

Alan Wills
University of Manchester
alan@cs.man.ac.uk

Fresco is a Smalltalk-based interactive environment supporting the specification and proven development of re-usable software components. These ‘capsules’ are deltas to the inheritance hierarchy, and form a more useful unit of designer-effort than class subhierarchies. Systems are built by composing capsules, which carry both specifications and code. The semantics of capsule composition is elucidated by examining the relationship between ‘type’ and ‘class’. Type-descriptions take the form of model-oriented specifications.
The principles discussed here can be applied to other object-oriented languages.

Keywords: Smalltalk, module, capsule, Fresco, Mural, subtype, inheritance, specification, program proof.

Published in Proc. ECOOP’91, Springer

1 Fresco

Fresco is an interactive environment supporting the evolutionary development of re-usable specified, proven software components. The prototype Fresco is based on Smalltalk: firstly, to preserve the evolutionary nature of Smalltalk programming, and to demonstrate that this is not incompatible with formal methods; and secondly, because of the availability of Mural, an interactive theorem prover’s assistant, written in Smalltalk, which can readily be integrated with the development environment.

Although Fresco in its current form extends Smalltalk, the principles should apply equally well to other object-oriented languages such as C++; and some investigation has been done in this direction. However, this paper concentrates on two aspects of the Smalltalk manifestation: namely, the type/proof system, and its support for Fresco’s novel ‘capsule’ system.

Fresco extends Smalltalk in two principal ways:

- Fresco systems are composed of re-usable units of software called ‘capsules’; Fresco attempts to guarantee that no mutual interference will occur between them.
- Fresco extends the Smalltalk language with a notation for describing behaviour, and provides tools for verifying the code of capsules.

The next section introduces the idea of capsules and explains their utility; following that, the type and proof system is described; and then we return to see how the semantics of composing systems from capsules is formulated in terms of the type system.

2 The formalised goodie

2.1 Formal Methods and OOP

Object-oriented programming makes possible a culture in which systems are rapidly built from widely-distributed and adapted components. Developers can build and sell or exchange components as well as complete systems; and can treat their software libraries as capital resources which they augment every time they write a new component. There are three good reasons why behavioural specifications are more necessary in this software engineering paradigm than in a more traditional one:

- With parts acquired from everywhere, the designer must be especially careful to have an unambiguous understanding of what each part is supposed to do, and some guarantee that it will indeed do that. If you have to test each part just as carefully as if you'd built it yourself, much of the advantage of re-usability is lost.
- Furthermore, if updated versions of a component are to be distributed and incorporated into systems which use it, the systems' designers must be able to distinguish those features of the component's behaviour which are incidental, from those which will be retained in future versions. (A sorting routine example: is it a guaranteed feature that items with equal keys retain their original order, or just an artifact of this version?)
- Lastly, polymorphic code generally requires the types with which it deals (or is instantiated) to conform to some restriction. It is insufficient to check that objects passed to a sorting routine all accept the binary operator '<': additionally, '<' must work like a proper ordering on them. In a closed system which is all written by one designer, it may be acceptable to document these restrictions informally or not at all; but where polymorphic code is to be distributed widely for use in conjunction with classes its designers have never conceived of, it is important both that the precise constraints on client-classes are documented, and that the code is guaranteed to work with any client class which conforms to those constraints. Otherwise, again, the client designers might as well build and test the distributed code for themselves.

These considerations argue for the desirability of stating the required behavioural characteristics of a software component in unambiguous language, and (better still) of checking each component against its requirements.

Full formal verification is difficult to achieve, and there are a number of alternative strategies such as symbolic execution and axiom-directed testing; but this paper proceeds under the assumption of the author's opinion that there a sensible mixture of formal verification (where easy or crucial) and informal justification — annotated with remarks like 'obvious' or 'Alan thinks this is OK' — (where hard and inessential); and that this style (dignified with the term 'rigorous proof' [Jones90]) is achievable with suitably friendly and well-integrated mechanical assistance. Proofs can be checked mechanically once generated, and so the re-use of any software implies the re-use of the associated proofs.

Such support is available in the form of the Mural interactive theorem-prover's environment, in which hierarchies of theory can be developed and proofs generated in the natural deduction system (as described in [Lindsay87]). Mural was written in Smalltalk as part of an earlier Manchester project[JJLM91]. It provides a generic proof system which can be adapted to a wide range of formal systems, and has manifestations as a stand-alone theory database, as a VDM tool, and most recently as part of the Fresco software development environment.

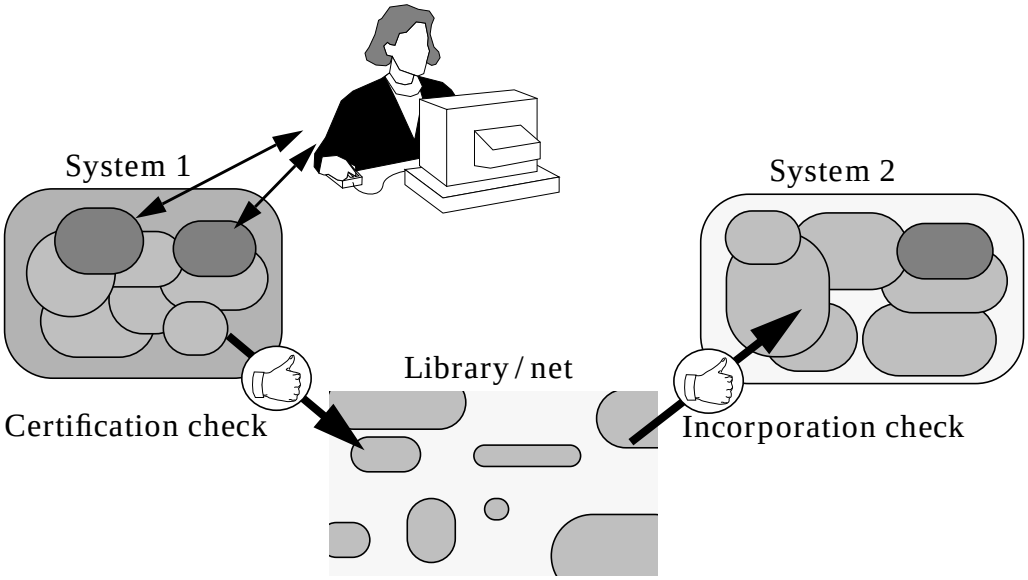


Figure 1. Fresco systems are compositions of re-usable capsules

2.2 Capsules

The units of distribution in the successful Smalltalk re-use culture are not classes, nor even groups of classes. A look at any 'goodies' library shows them to be mixtures of new classes, new methods for existing classes, and new implementations of existing classes and methods. (In Smalltalk, classes and methods are updated and compiled dynamically into the running system.) In goodies-lib@cs.man.ac.uk, 73% of the files modify existing classes, and 44% define no new classes. Each programmer's efforts build upon those of one or more predecessors by improvement and extension. Fresco formalises this mechanism.

If this notion of deltas as units of designer-effort seems a little strange, consider this scenario. Class A uses class B extensively, and sometimes passes B-instances back to its own clients. I design class C, which uses A; but C needs B to perform some extra function, used whenever B-instances are passed back from A. Ideally, I should design a B′ which inherits from B. But then I have to design an A′ which is all the same as A, except that it calls upon B′ instead of B. If A has been designed with sufficient foresight, then this will be easy; but more likely, it will be a pain! What I really want to do is just to add the extra function to B — more economical and less error-prone. More generally, many of the real-life examples of redefinition are connected in some way with improving the inheritability of a class, or broadening its functionality. Others are concerned with improving the performance (so that all clients get the benefit, not just those who know about the subclass); and most of the rest, with enhancing user-interaction without altering the procedural interface.

Functional units and their hierarchies are good for integrating into one structure all the diverse functions which can be created by a single designer [team] while the hierarchy remains under that designer's control; additional requirements may trigger a restructuring. But when we consider design effectively undertaken by many designers between which there is only a one-way flow of information, then the transmissible units of design-effort must be not functional units, but changes to their definitions. But it is important that when a system imports such deltas from diverse sources, they shouldn't invalidate each other: each should be able to change the implementation of what went before, and should be able to enrich any part of the system's behaviour, but not to alter (or delete!) the functional specification of existing behaviour, which other parts might depend on.

Fresco supports the specification and rigorous development of software capsules. A capsule

contains code, specifications, and proofs, and systems are built by composing capsules. The mechanism has the potential to guarantee that each capsule functions as its author intended, without interference from others: although the functions a capsule provides can subsequently be extended or improved, the properties its clients rely upon will never be invalidated.

Part of the system's operation depends on restricting the ability of a capsule to override existing definitions, to those belonging to capsules on which it has a documented dependence: this by itself can help to reduce the likelihood of clashes. Whilst the full benefit depends on the (admittedly theoretical) employment of fully formal proofs, greater reliability is nevertheless obtained by using specifications with more or less 'rigorous' proofs. Even where proofs are completely informal, the system highlights correspondences between specification and code which should be rechecked whenever anything is altered.

A capsule may be created in any order: code first or specifications first. Fresco generates appropriate proof obligations wherever the consistency of the code and specifications cannot be verified automatically. Before the capsule may be exported for distribution to other designers, Fresco performs a 'certification check', that all the proofs have been completed, and are consistent with the definitions (see Figure 1). A complementary 'incorporation check' ensures that imported capsules (i) only alter the code of capsules they claim to know about and (ii) have internally consistent proofs (even if partly informal ones) and hence, hopefully, code that conforms to their specifications.

These mechanical features are not the main topic of this paper: rather, we concentrate on the notions of class, type, and conformance, and on the semantics of composing capsules into systems.

The next section introduces Fresco's type system and outlines how it fits into the proof system. We will ultimately return to capsules and explain their composition into systems, in the light of the type system.

3 Classes and types

3.1 Type/Class Definitions

'Class' and 'type' are distinct ideas in Fresco. Classes prescribe implementations of types. Types describe behaviour, visible as the object's response to messages and the constraints which apply to messages it may be sent. An object is an instance of only one class, but may belong to many types: any class implements many types.

Types are used to document constraints on variables or parameters. It's important to notice that a Fresco type can constrain more strongly than the machine-checkable types of conventional languages, since it requires that a member-object conform to a particular model of behaviour in response to a particular set of messages (procedure calls); compare this with C++ for example, where type membership imposes the requirement that an object should be able to respond to a given set of messages, but doesn't stipulate how.

Despite the distinction between class and type, type and class definitions are nevertheless interwoven, for convenience, into a single all-purpose piece of syntax, the type/class definition (TCD). Fresco has TCDs instead of classes, and they are realised by adding extra information to Smalltalk's classes. Smalltalk's interpreter treats a TCD exactly as a class, ignoring the type information; whilst typing assertions and type-building expressions ignore the class aspects of a TCD.

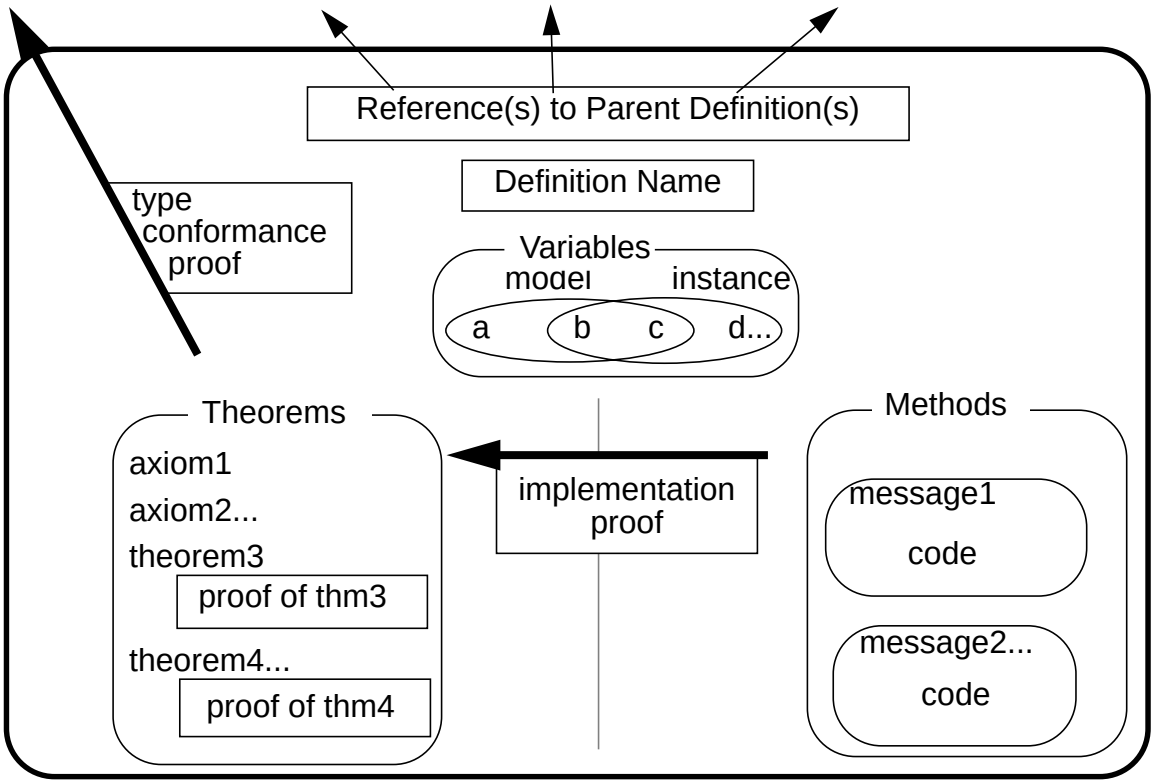


Figure 2. Components of a Fresco combined type/class definition (TCD)

The following headings summarise classes, types and their relationships. Figure 2 illustrates the components of a Fresco TCD, and its relationships to other TCDs. The relationships of inheritance and conformance are the most significant.

The composite definitions are perhaps more easily explained by considering the two roles separately.

Class definitions

The essential components of a class definition in Fresco are unchanged from Smalltalk:

- Name of class
- Identity of parent class definition(s)
- Instance variables
- Mapping from selectors (operation signatures) to methods (operation code bodies)

Type definitions

Types are used to characterise the properties of objects which may be assigned to variables or parameters; and to characterise the properties of classes which may be used in conjunction with polymorphic code.

Fresco type definitions take the form of axiomatic specifications, with these components:

- Name of type
- Identity of parent type definition(s) (**not** supertype, necessarily)
- Model variables
- Theorems (axioms and derived theorems)

The type encompasses all those objects which behave according to the theorems. (Axioms are those which define the type; other theorems may be proven from the axioms, for the convenience of clients.) The theorems state invariant properties of the model, and describe the effects of messages (operation calls) on the model, using pre- and post-conditions. A model need not be used — it is possible for the theorems just to interrelate operations visible at the client interface — but experience in the formal methods culture suggests that model-based specifications are easier to read and write for all but the most fundamental types.

We write c∈ T iff c behaves according to T's theory: that is, the visible behaviour of c (how it responds to operations) satisfies T's axioms. If c's own internal components are invisible or different from T's, then you can still decide whether c behaves according to T's theory: generate (in the abstract!) all the theorems derivable from T's axioms; discard all those which mention the internal variables, leaving those which just mention external operations (like an algebraic spec); checking that all possible sequences of operations on c conform to those. In practice, there is an easier and (mostly) equivalent way, described below under 'reification'.

3.2 Theorems

Theorems are the basis of all the behavioural description in Fresco. The general form (which is developed from Cline and Lea's work on Annotated C++ [CL90]) is:

{ |vars| theorems ⊢ precondition :– postcondition } [code] .

All parts may be omitted except postcondition.

vars are metavariables, with scope throughout the theorem, including the code. When the theorem is applied in a given context, each metavariable may be instantiated to match a particular expression. Unmatched metavariables effectively work like universally quantified variables.

Initial theorems preceding ⊢ are hypotheses: the rest of the theorem is valid iff the hypotheses can be proved. This kind of theorem can function as a proof rule.

The theorem states that if precondition is true before the execution of code, then postcondition will be true after it.

If 'precondition :-' is missing, the theorem is an invariant, true after the code if it is true before it; if code is missing, it is a universal invariant, true all the time.

precondition and postcondition are expressions. In postcondition, an overlined variable represents a copy of the same variable as it was before execution. (The copy is guaranteed not to be subject to any side effects of the code! The problems of interpretation and aliasing this presents are not the topic of this paper, but are the subject of ongoing work.)

code may be an expression or a sequence of statements. (In Smalltalk, conditionals and loops are expressions.) It may also be another theorem.

If code is omitted but ':-' is present, then the theorem represents any code which satisfies the theorem. This form may be used as a stand-in for as-yet-undeveloped code.

The syntax of the predicate parts of the theorems is adapted from that of the programming language. It may include inexecutable constructs such as quantifiers (unless the intention of the user is to employ the theorems as a debugging aid: which is not the assumption here). (Readers unfamiliar with Smalltalk should be aware of the postfix and mixfix syntax of its operators.)

A theorem may appear in any of four principal roles:

- As a ‘specification statement’, within the code of a method as an annotation and possibly a debugging aid. A theorem is executed by executing its code-part. It asserts that the code satisfies the pre/postcondition part. (Hypotheses are not used in this context.) A theorem can be proven by the use of rules which analyse its code, so that the code of a method can be proven stepwise — as fully explored in [Morgan90]. An outermost theorem of a method can be used to prove axioms of its class’s home type. E.g.:

  { x∈ Real ⊢ :- (y>0) ∧ (y∈ Real) } [y:= x abs].
  { y∈ Real ⊢ x > 0 :– z∗z = x } [z:= y sqrt] .

- As a step in a proof, following the style of [JJLM91]. Theorems may be used as proof rules; and are themselves proven by proving the conclusion of the theorem (that is, the parts left when hypotheses are removed) in a local context in which the hypotheses are assumed. Such proofs typically look like this (though regrettably, on a much larger and commensurately tedious scale):

  h x∈ Real
  1 {x > 0 :– ↑∗↑ = x} [x sqrt] . **by** defn-sqrt(x) **from** h
  c {x > 0 :– y∗y = x} [y:= x sqrt] . **by** assign-↑(y) **from** 1

  (↑ refers, in a theorem, to the value yielded by the code.)

- As a proof obligation to justify claims of correct implementation, conformance, etc. For example, if we wish to claim that members of Colour can be sorted using the polymorphic class SortedCollection, then we must satisfy the latter’s requirement that objects it deals with should be members of Ordered, whose axioms are (say) OA_*i*:

  { {x∈ Colour ⊢ OA_1}. {x∈ Colour ⊢ OA_2} ... ⊢ Colour ≤ Ordered}.

- As an axiom or derived theorem of a type. If we want to prove the assertion x∈ T where T is some type, then we must prove that all of T’s axioms are satisfied by x. Conversely, if we know that x∈ T, then we can use T’s theorems as rules to prove things about x.

A body of axioms, together with all the theorems that can be derived from them, and usually some locally-declared variables, is called a theory. A theorem is always stated in the context of some theory. A type is defined by its theory (and hence each class with a home type is specified by a theory); and there is also a background theory, inherited by every other, which includes the usual rules of predicate calculus, together with the behaviour of the Smalltalk kernel; its theorems include, as examples:

{ |P R b S1 S2 | b∈ Bool, {P :- R } [b ifTrue: S1 ifFalse: S2]
⊢ P :- R } [b ifFalse: S2 ifTrue: S1] . "ifTrue:ifFalse: reversal"

{|P M R S1 S2| {P:-M} [S1]. {M:-R} [S2] ⊢ P :- R} [S1. S2] .

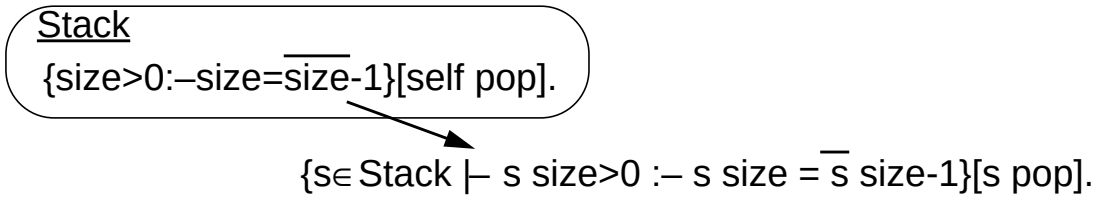
{|P M R S1 S2| P :- R} [{P:-M}[S1]. {M :-R}[S2]]. "corollary of previous"

The operation specialisation axiom is especially important:

{ | P P1 R R1 S |
{P ⊢ P1}. {R1 ⊢ R}.
{P1 :- R1} [S]
⊢ P :- R } [S].

In the theory of a type, there are some implicit variables and axioms. ‘∈’ is the type membership relation, and in the context of any one type T, it is automatically axiomatic that self∈ T. If a theory is extracted from the context in which it is stated, the theorems it depends on must be taken with it as extra hypotheses. In particular, when some theorem of T is used in a proof of a

client, self∈ T must be added as a hypothesis (and then self must be substituted by some other metavariable, to avoid naming conflicts).



3.3 Relationships between types and classes

Classes implement types

Class and type definitions may be mixed into one Type/Class Definition. The implication is that the class is intended to implement its 'home' type. This can be verified by proving that each axiom is satisfied by the code of the methods – an 'implementation proof'. For axioms whose code part is in the form self message, the code of the appropriate method is unfolded, and a proof by decomposition is done in a style after [Jones90] or [Morgan90]. Additionally, any invariant (axiom without precondition or code parts) may be assumed in conjunction with the precondition, and must be proven as a postcondition of each method.

This arrangement is similar to the way in which Eiffel [Meyer88] classes possess invariants and pre/postconditions: in Fresco, these functions are performed by the axioms.

(Since the technique is to prove the axioms true of the code, 'axiom' might seem a misnomer. However, it is appropriate in the sense that clients of the class assume the axioms to be valid, whilst it is an internal affair of the class's to get its code to fulfil the axioms.)

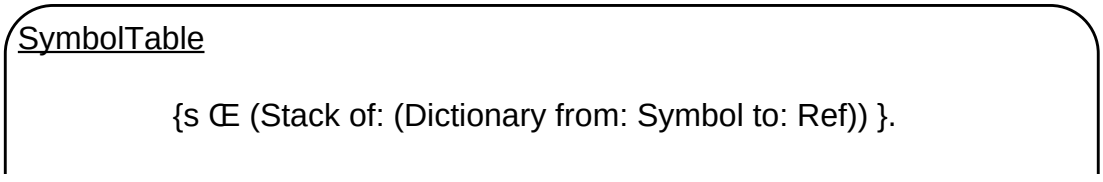
Whilst it seems good practice in general to restrict each axiom to determining the behaviour of one operation, there may be several axioms applying to one operation. This may arise through inheritance or capsule composition, or just because it's convenient. In that case, the axioms must each be proven against the code of the operation; or it might help to invent a lemma from which the 'axioms' follow, and prove the lemma against the operation.

It is possible for the axioms of a type to contradict each other — in which case, there can be no implementation. The only protection against this is the unprovability of any code which might be written for such a type.

Since theorems work as proof rules, new rules may be introduced with each class; so that its messages have the same status as basic linguistic constructs. Contrast this with the conventional proof system, in which there is a fixed set of rules for the language, some of which deal with procedure calls in a general and rather clumsy manner.

Example

A TCD SymbolTable is intended for use by compilers of block-structured languages, and maps identifiers to some form of reference. Identifiers may be declared within nested blocks of the language, so SymbolTable is modelled as a stack of dictionaries, the range elements of which are References:



(Every member of SymbolTable has a component s. x∈(Stack of: T) is defined elsewhere to be equivalent to (x∈Stack) ∧ ((xi in: x) ⇒ (xi∈T)); and similarly for Dictionary.)

There are four operations, for entering and leaving nesting levels, for adding an identifier at the current level of nesting, and for finding the most deeply-nested current declaration of an identifier. Each of these 'advertised' operations is mentioned in the code part of one or more axioms (as opposed to model-components like s, which only occur within the braces). To give two axioms, as examples:

```
{ id∈Symbol :– (s top at: id) = ref } [self declare: id with: ref].
{ (s size > 0) :–  s = s~ tail}  [self leave].
```

(The operation declare:with: is not guaranteed to work unless its first argument is a Symbol; its result is that the Dictionary at the top of the Stack is now such that interrogating it about id yields ref. The operation leave works only if the stack is not empty, and its result is to reduce the stack to its former tail.)

(In this paper we ignore questions of framing — how to stipulate that the other members of the structure remain unaltered.)

Inheritance

Inheritance is a relationship which a designer may prescribe between definitions; it doesn't imply conformance between the behaviours described, nor vice versa. Variables and theorems are inherited from parent definitions; and methods are inherited, but (as in Smalltalk) may be overridden in child definitions. Theorems cannot be overridden in children.

There is multiple inheritance in Fresco. Synonymous variables inherited from different parents are identified; label-clashes amongst theorems are resolved by qualifying them with the names of the parents from which they come; synonymous methods are disallowed unless the child definition provides an overriding method.

The terms 'parent' and 'ancestor' will be used instead of 'superclass' here, to minimise traditional confusion between inheritance (the carrying of features from one definition to another) and conformance (where one behavioural definition satisfies another) [CHC90].

Conformance

Type C conforms to type A, written C≤A, iff ∀c · c∈C ⇒ c∈A.

The conformance relation between types is used to determine whether member-objects of C may be supplied wherever A is expected. In turn, it can therefore be used to determine: whether one type (or class) correctly implements another; whether a type fulfills the requirements of a polymorphic piece of software; and whether a proposed modification to a class will produce a substitutable variant.

There is a component of each TCD, in which intended conformance to other type(s) may be recorded. An appropriate proof should also be recorded there; Fresco highlights the absence of one: literally so, in any screen display of the type, and also in the sense that a certification check will fail on an absent proof. Any change to the type at either end of the conformance relation will cause a similar behaviour until the proof is at least re-affirmed. These checks are beneficial even if the proof is completely informal, since the designer is forced to re-consider any assumption or justification which may no longer be valid.

To prove C≤A, we only have to prove that all the theorems of A hold in C: that is, that A's axioms can be derived as theorems from C's axioms. This means of course that C must provide for at least the same set of messages as A; beyond that, there are three interesting cases:

- C's model is different from A's. Some translation has to be done, in the form of a *retrieve axiom*, which interlinks the two models: the proof is made feasible by adding this extra axiom to C. The operation specialisation axiom above is crucial to such proofs, since most of the axioms specify the effects of messages. We also need to prove 'adequacy' – that there are sufficiently many states of C to represent A.
- A is an ancestor of C. In the case, the retrieval is trivial, since it just involves dropping the extra variables from the model. To ensure adequacy, C must avoid constraining variables inherited from A.
- A is not defined with the aid of a model. Fresco uses a loose interpretation in which there is no adequacy proof in this case. This is appropriate for specifications of individual properties.

Conformance and inheritance

A TCD's complement of axioms includes those inherited from others; and any methods it possesses must be proven against the inherited axioms as well as its own. There may be more than one axiom relating to each operation. Conversely, inherited methods must in general be proven against the class's own axioms, even if they have been proven against axioms in their own TCD.

In practice, Fresco allows only certain combinations of conformance and inheritance — the others seem unuseful or confusing. They are:

- *Reification*—conformance claimed between TCDs otherwise unrelated.
- *Nonconformant inheritance*—in which the variables and methods of a TCD may be inherited without at the same time inheriting its theory.
- *Conformant inheritance*—in which a TCD is claimed to conform to a parent, which will have axioms and may or may not have methods.

(The last two correspond to the two kinds of inheritance in C++: private and public.)

In conformant inheritance from a TCD which has both theory and methods, any inherited axiom need not be re-proven unless it relates to a method which is overridden or newly-defined in the child.

3.4 Encapsulation and Reification

'Encapsulation' is the idea that the clients of a unit of software design should depend only on its published interface, not on its innards: the knock-on effects of a change of implementation stop at the unit's boundaries, provided the interface description remains true. In OOP terms, this means that it is none of a client's business to use a class's internal functions, or to see internal data structures. This has sometimes been seen as prohibiting a model-oriented approach to specification, since such specifications describe externally visible behaviour in terms of their effects on an internal state.

This is true if the model data are constrained to be the same as the implementation variables: but no such restriction is necessary or desirable. For example, a dictionary may be modelled in TCD Dictionary as a set of key-value pairs with a uniqueness constraint on the keys; but it could be implemented as a tree in TCD TreeDict. The latter would carry a claim and proof that TreeDict≤Dictionary. Dictionary is advertised as the interface specification of TreeDict; and

though Dictionary has its own model, it gives no insight to TreeDict's internal workings. Dictionary might have no executable code of its own; or it might, after the conventional style of Smalltalk abstract superclasses, offer additional operations built onto the specified interface. Thus TreeDict preserves its encapsulation by making its public interface the claim to implement another TCD.

'Data reification' — implementing one model with a different one — is a valuable technique, with considerable respectability in the literature of formal methods[Jones90]. The most appropriate model for human readers is by no means often the most appropriate for implementation, and performance cannot always be improved merely by adding fields (as in conventional inheritance).

An alternative approach is a purely algebraic style, in which the axioms interrelate only the externally visible operations. This works well for the most fundamental types (and indeed is the only way to specify them), but is difficult to use for more complex specifications.

It is common for a design to contain several stages of reification, each stage being a re-modelling of the preceding one.

Reification example

```
TableDict
        {(dict ∈ (Dictionary from: Symbol to: (Dictionary from: BlockId to: Ref)))
        ∧ (blockCount ∈ BlockId)
        ∧ (currentBlocks ∈ (Stack of: BlockId)) }.
```

This improves the efficiency of SymbolTable. There is one dictionary, in which all the current identifiers can be rapidly looked up; each of them has a stack of current (and some past) references, each associated with a block number so that outdated entries can be distinguished from current ones. enter and leave number the blocks, and keep track of which are current. The retrieval to SymbolTable is:

```
{ | i name |
        (0 < i) ∧ (i ≤ currentBlocks size) ⊢
        ((name in: (s at: i) dom) ⇔ ((currentBlocks at: i) in: (dict at: name) dom)) ∧
        (((s at: i) at: name) = ((dict at: n) at: (currentBlocks at: i))) }.
```

(For any index i to the stacks s (in SymbolTable) and currentBlocks (in TableDict), every name is found in the domain of the ith dictionary of s iff the current nesting block's id is in the domain of the subdictionary of dict at name; ....)

This structure is:

- obviously more difficult to understand than SymbolTable
- not just an extension of the SymbolTable
- unsuitable as an interface specification for the symbol table, since there are many other ways in which SymbolTable could be implemented
- not the final stage of reification, since we still have to decide which of many possible implementations of Dictionary to choose for its two occurrences
- clearly more appropriate for implementation than any extension of SymbolTable.

TCD conjunction

When two axioms {P1:–R1}[s], {P2:–R2}[s] apply to one code fragment, the effect is to weaken the precondition and strengthen the postcondition. It follows from the specialisation axiom that

$$\{(P1 \vee P2) :- (\overline{P1} \Rightarrow R1) \wedge (\grave{}\overline{P2} \Rightarrow R2)\}[s] \vdash \{P1 :- R1\}[s]$$

and the same for P2:–R2. If the preconditions are disjoint, the effect is to stipulate independent domains in which s should work; if they overlap, then $\{P1 \wedge P2 :- R1 \wedge R2\}[s]$ will apply. Any object conforming to the conjunction $\{(P1 \vee P2) :- (\overline{P1} \Rightarrow R1) \wedge (\grave{}\overline{P2} \Rightarrow R2)\}[s]$ thus conforms to the two originals. However, it is normally only necessary to consider the original axioms, rather than dealing with this conjunction explicitly.

This axiom conjunction occurs when axioms from an inheriting TCD and its parent(s) apply to the same method; or when the designer chooses to separate different concerns.

This can be extended to work for whole TCDs. The conjunction T1&T2 of two TCDs is formed by merging the axiom-sets and variable-sets; and by overriding any methods in T1 with those of the same name in T2. So if a TCD is a tuple <vars, axioms, methods>, where methods is a map from message selectors to method bodies and $\ominus$ is symmetric set difference,

$$\begin{aligned} &\langle vars1, axioms1, m1\rangle \,\&\, \langle vars2, axioms2, m2\rangle = \\ &\quad \langle vars1 \cup vars2,\ axioms1 \cup axioms2, \\ &\qquad \{sel \rightarrow (m1 \cup m2)(sel) \mid sel \in (dom(m1) \ominus dom(m2))\} \\ &\quad \cup \quad \{sel \rightarrow m2 \mid sel \in (dom(m1) \cap dom(m2))\}\rangle \end{aligned}$$

The conjunction is valid if the proof obligations can be satisfied of

- conservative extension

  $\forall a \in A\ b \in B \cdot \exists ab \in (A\&B) \cdot ab\big|_A = a \ \wedge\ ab\big|_B = b$

  (where $ab\big|_A$ means removing the components which don't belong to A).
- correct implementation – satisfaction of the axioms by any methods in the result.

We claim that a valid composition A&B satisfies

$$\forall ab \in (A\&B) \cdot ab \in A \ \wedge\ ab \in B$$

Notice that TCD conjunction is symmetrical in its type components, and asymmetrical in the code — B's methods override A's.

The type defined by a conformant inheritor C of A is A&C.

Another interesting use of & is to split up specifications of operations for descriptive purposes: for example, the main and exceptional behaviour of an operation can be written separately.

Type constructors

Fresco has a few built-in type constructors, listed here with examples:

- union: List = Cons | EmptyList
- product: ListPair = List × List
- functions: ListDyadicInjection = ListPair → List
  cons ∈ ListDyadicInjection
- filter: ShortList = List ! [x | x length ≤ 5]
  (all members x of List such that ...)

Generic types, written as functions over types:

- T set = (Set ! [s | ∀ i∈s · i ∈ T])
- s∈Set ∧ (∀ i∈s · i ∈ T) ⊢ s ∈ (Set of: T)
- (Map from: T1 to: T2) = (Map ! [m | ∀ (d,r)∈m · d∈T1 ∧ r∈T2])
- T ≤ Ordered
  ⊢ sc∈SortedCollection ∧ (∀ i∈(1.. sc length) · (sc at: i) ∈T)
  ⊢ sc∈(SortedCollection of: T)

The user may define these arbitrarily.

The principles of conformance proof may readily be extended to cover these constructions.

Type checking

Although there is no typechecking in Smalltalk, nor at present in Fresco, types are nevertheless a useful tool: c∈T abbreviates the restatement of all T's axioms. There are no type constraints on parameters or variables, but we can nevertheless state that an axiom's conclusion depends on the assumption that c∈T: if the axiom is used as an invariant and c is an instance or model variable, then that is equivalent to stating the variable's type; all we lack is an automatic means to check this – the typing proposition just has to be proved like any other. (It could be argued that, since program development with Fresco is an interactive process, we no longer suffer such a strong imperative to separate the automatically-verifiable constraints (like type-checking) from the proofs which need the human touch.)

Typechecking has been added to Smalltalk [JGZ88] but there as in most languages, the idea of conformance is limited. Whilst the compiler can check that T′ provides all the operations of T, it cannot check that they behave substitutably. Moreover, most OO typing schemes overlook the important 'reification' case in which one type conforms to another without inheriting its definition. The typechecker of POOL [AvdL90] is a step in that direction: it takes the names of informally-defined properties as clues to what is required in a conforming type.

In Fresco, conformance means substitutability, with or without inheritance. Of course, a solid guarantee of conformance would require a watertight proof, whereas informal justifications are allowed in Fresco. Nevertheless, the experience of the formal methods tradition is that the obligation to supply even informal proofs brings about a measurable increase in reliability.

4 Capsules

All Fresco software development work — specification, coding, proof, documentation — is done within the context of some capsule. A designer may develop several at once within the same system, but has to switch consciously between them: each corresponds to a separate 'desktop'. Once developed, the designer can ask Fresco to certify the capsule: that is, to check that the proof obligations are all up-to-date and have complete proofs. A certified capsule can then be incorporated into another system.

Each capsule has a name which is unique worldwide: the full identification includes date and hostid of origin, and author's name etc. are included in the 'header' documentation. Each builds on the work embodied in other capsules, and a capsule's attributes include the names of its prerequisite capsules. A capsule cannot be incorporated into a system unless its prerequisites are already there. The prerequisite graph is acyclic and directed; capsules are not functional modules, but modules of programmer effort: if two modules are interdependent, then they should be defined as separate TCDs within the same capsule; capsules' dependencies are

unidirectional.

During development, Fresco ensures that the designer does not use (or inherit from) anything defined by another capsule which is not a prerequisite. As far as TCDs and global variables are concerned, this is just a question of tracing the definitions of names: every definition in Fresco is associated with a particular capsule. But for messages, this can't be done with complete certainty until an attempt to construct a proof, which must refer to the definitions of operations in particular types.

On incorporation into another system, Fresco checks that the definitions given by the incoming capsule do not clash with those of other capsules which are not its prerequisites. A renaming scheme can be invented which circumvents some of the problems, where a new definition accidentally has the same name as something else. But in the case where two cousin capsules (with a common prerequisite, but neither prerequisite of the other) try to redefine the same item in different ways, then they can only be declared incompatible and cannot both become part of the same system.

A capsule may only define new TCDs and conformant augmentations of existing ones. The TCDs in a capsule are therefore composed using '&' with the ones already existing in the system (which should come from prerequisites); so that the new code implements the old specification as well as the extension. (Figure 4.)

Once certified and published, a capsule cannot in general be modified (without renaming it); but a new version may be issued if it conforms to the old one. An extension to the naming scheme encodes the version history (branches are allowed, of course: improvements may be made by diverse authors), and prerequisites must be quoted with name and version. Then any later version will be a satisfactory substitute.

4.1 Capsule contents and composition

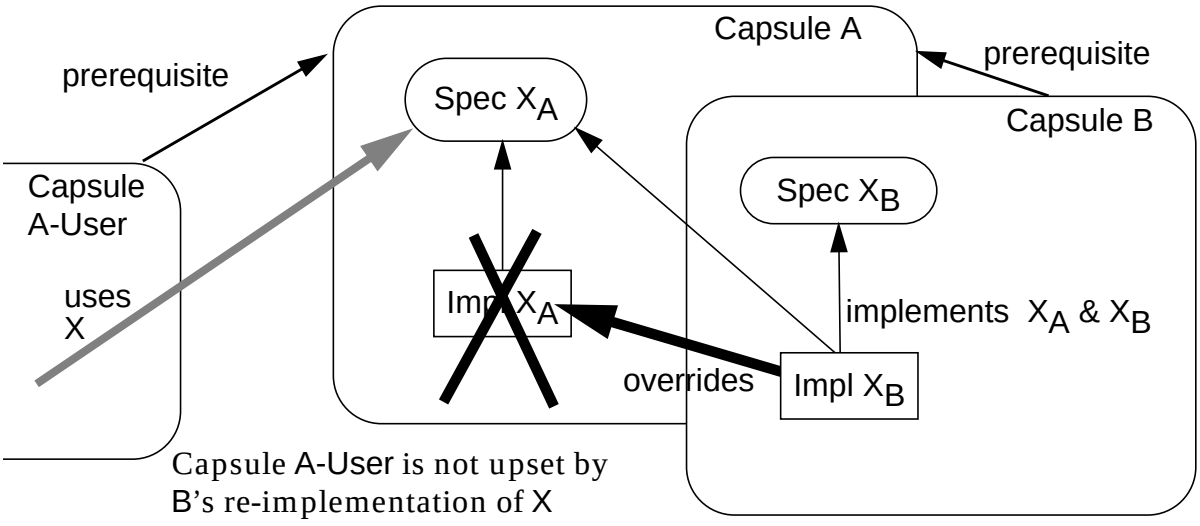
A capsule is a tuple ⟨name, version, prerequisites, definitions⟩.

Name, **version** and **prerequisites** have been covered above.

Definitions includes all TCDs, together with global-variable definitions.

TCDs include variables, methods, theorems, proofs, conformance claims.

Figure 4. Capsule composition conjoins specs and overrides implementations



A Fresco system is a tuple ⟨capsules, definitions, run-time-stuff⟩.

Capsules is a list of the capsules the system has incorporated. All definitions can be attributed to a particular capsule. Every system has a Kernel capsule, which contains all the standard-issue classes and globals.

Run-time-stuff is the heap, stack, interpreter state, and so on, which depends on the **code** in just the same way as it does in ordinary Smalltalk.

So the definitions in a system are determined by its capsules, and by the order in which they were incorporated, which in turn is determined by the prerequisite graph. Each capsule's incorporation produces a resultant system which is the & of the old system and the capsule.

For a system s and capsule c (where ⌢ is concatenation),

$$c.prerequisites \in s.capsules \vdash s \,\&\, c =$$
$$\langle \quad s.capsules \frown c,\ s.defs \,\&\, c.defs \ \rangle$$

which adds the new capsule to the list, conservatively extends the types, overwrites method definitions and adds fields to classes, replaces reification claims by new ones (by target), and unifies theories such that all the theorems of the old system are still true in the new.

The term of principal interest is s.defs & c.defs, which is defined thus, if defs is a mapping: Name → TCD:

$$s.defs \,\&\, c.defs \triangleq$$
$$\{n \to (s.defs \cup c.defs)(n) \mid n \in (dom\ s.defs \ominus dom\ c.defs)\} \cup$$
$$\{n \to s.defs(n) \,\&\, c.defs(n) \mid n \in (dom\ s.defs \cap dom\ c.defs)\}$$

so that the new capsule may contain new TCDs or extend existing ones with &. Extended TCDs should only include those belonging to prerequisites: so the author of the capsule must already have proved the validity of the composition; and will also have proved that the resultant specifications are met by any code which is added to the capsule in the future.

5 Continuing work

5.1 Framing

The examples of theorems shown here omit 'framing': any statement of what parts of the system's state may be altered by the code. In most model-oriented specification systems, the frame may be simply a set of variables, or it is a well-defined subcomponent of the data represented in a variable. That would be inadequate for Fresco:

- The heavy aliasing in Smalltalk means that it isn't so easy to divide up the state into subcomponents, some of which are writable and some of which are not: two subcomponents you thought were different might turn out to be the same, accessed through different strings of pointers.
- In a modular system, what is changed by an operation is partly the responsibility of server modules, so it is insufficient just to name a given part of the whole state.

A way of tackling this is the effects system [LG88]. In the realisation planned for Fresco, each

class has a *demesne*, which identifies the set of objects which help it represent the type it is intended to represent. Normally, the demesne would be self + the demesnes of the components — though 'knows-about' pointers and caches would be excluded. We introduce a special assertion which can be appended to a postcondition, effect(someDemesne), together with rules for reasoning about such effects. This system should be sufficient to separate out all those problems associated with the fact that we are dealing with pointers rather than pure values.

5.2 Using programming language for specification

It is desirable to integrate the programming and specification languages; but it is difficult to understand the meaning of a postcondition or a line in a proof which itself has a side-effect on the system state.

The effects system gives us a way of reasoning about the purity of an expression: if we can prove that its effect is the empty demesne, then we are allowed to use it inside a theorem. So for example, it should be possible to discover which operations confine their effects to the demesnes of the receiver objects to which they are applied; so that such an operation followed by a 'deep' copy would be pure.

6 Summary and current status

The notion of conformance (= 'subtyping', but with the caveat that we aren't just referring to signatures) is important in the determination of whether one definition implements another, particularly in these contexts:

- To facilitate clear description, disjoint from implementation concerns, for re-users.
- Polymorphic code must define the properties of objects it is capable of working with.
- Adding an upgrade or delta to any piece of software should leave it still able to serve previous clients as it did before: it should satisfy the same specifications.

The last point is perhaps less familiar, and applies to capsules, which are monotonic deltas to system definitions; and which this paper argues form better units of re-use than classes do.

Composition of specifications and software must be guaranteed conformant in certain cases:

- Where a type is defined as the child of another, the intention is often that its members should be substitutable for its parents' members. Its own definition and the parent's must therefore be composed conformantly. The same applies with multiple parents.
- Polymorphic code may have several constraints that objects it is applied to must satisfy. The notion of a conformant composition of types is therefore useful.
- Composing capsules to make systems should not result in mutual interference between the constituents, which must therefore be conformantly composed.

Fresco type-definitions are in general model-oriented, which makes them easier to write and read. Encapsulation is not compromised by this approach if the technique of reification — implementing one model with a completely different one — is adopted.

Type and class definitions are combined, in Fresco, into one syntactical unit, the TCD. This provides for:

- separate type and class definitions where required.

- a ‘home’ type for a class, in which its instance variables form the model: implementation is verified by checking that each method satisfies all the applicable theorems.
- the conventional ‘abstract class’ style, if preferred.

Type descriptions are made up of theorems about the effects of code fragments on data. Theorems function as proof rules: as more TCDs are added, the body of rules increases. Theorems can act as ‘specification statements’, and code is proven by decomposition. Theorems and proofs are written in an extensible pure subset of the Smalltalk language.

The conformant composition of capsules to build systems has been described in terms of conformant composition of the constituent TCDs.

Fresco supports conformant composition of capsules and TCDs, and code decomposition proofs, with a combination of mechanical checks and rigorous proof.

The current Fresco system is built on Smalltalk, whilst the principles may also be made to work in other languages. It currently supports capsules and specification annotations; but proof obligation generation and full linkage to mural are yet to be implemented. A mechanical typechecker would be beneficial. Current work includes the method of treating framing, aliasing, and the more reliable integration of the Smalltalk and specification notations.

Acknowledgments

Warmest thanks are due to Mary Brennan, John FitzGerald, Cliff Jones, the Nationwide Anglia Building Society, and the referees of this paper, for their diverse kinds of support.

7 References

[AvdL90] Pierre America and Frank van der Linden ‘A Parallel Object Oriented Language with Inheritance and Subtyping’ ECOOP 90

[CL90] Marshall Cline and Doug Lea: ‘The Behaviour of C++ Classes’. Proceedings, Symp OOP Emphasizing Practical Applications, Marist Coll., Sept 90

[CHC90] William Cook, Walter Hill, Peter Canning ‘Inheritance is not subtyping’ ACM ToPLAS 1990 pp125–135

[JGZ88] Ralph Johnson, Justin Graver, Lawrence Zurawski ‘TS: an optimizing compiler for Smalltalk’ OOPSLA’88

[Jones90] Cliff Jones: ‘Systematic software construction using VDM’ (PHI, 2nd ed. 1990)

[JJLM91] C.B.Jones, K.D.Jones, P.A.Lindsay, R.C.Moore: ‘Mural: a formal development support system’ Springer Verlag, 1991

[Lindsay87] Peter Lindsay: Logical frames for interactive theorem proving. TR: UMCS 87-12-7, Dept of Computer Science, University of Manchester, UK 1987

[LG88] John M Lucassen and David K. Gifford: ‘Polymorphic effect systems’ Proc 15th ACM Symp Principles of Programming Languages Jan 88 pp47–57

[Meyer88] Bertrand Meyer: ‘Object-oriented software construction’ (PHI 88)
[Morgan90] Carroll Morgan: ‘Programming from Specifications’ (PHI 1990)
[Wegner90] Peter Wegner ‘Concepts and paradigms of OOP’ *OOPS Messenger* 1(1) Aug90 [ACM]