A Formal Approach to Software Component Specification

Kung-Kiu Lau
Department of Computer Science
University of Manchester
Manchester M13 9PL
United Kingdom
kung-kiu@cs.man.ac.uk

Mario Ornaghi
Dip. di Scienze dell'Informazione
Universita' degli studi di Milano
Via Comelico 39/41, 20135 Milano
Italy
ornaghi@dsi.unimi.it

Abstract

There is a general consensus that the paradigm shift to component-based software development should be accompanied by a corresponding paradigm shift in the underlying approach to specification and reasoning. Work in modular specification and verification has shown the way, and following its lead, in this position paper, we outline our approach to specifying and reasoning about components, which uses a novel notion of correctness.

1 What is this paper about?

As the title suggests, this paper is about an approach to formal specification of software components. The purpose of such an approach is to allow formal reasoning about components. The ultimate goal of Component-based Software Development (CBD) is *third-party assembly*. To achieve this, it is necessary to be able to specify components in such a way that we can reason about their construction and composition, and correctness thereof, *a priori*. Work in modular specification and verification, e.g. [9, 14] has shown the way, and our approach follows its lead. However, our approach is novel and hence different in the way we define correctness. In this paper, we will discuss how we specify components, and in particular how we define and reason about correctness, and why this is useful for CBD.

2 Specifying Components

Ideally components should be *black boxes,* in order that users can (re)use them without knowing the details of their innards. In other words, the *interface* of a component should provide *all* the information that users need. Moreover, this information should be the *only* information that they need. Consequently, the interface of a component should be the *only* point of access to the component. It should therefore contain all the information that users need to know about the component's *operations*, i.e. what its code does, and its *context dependencies*, i.e. how and where the component can be deployed. The code, on the other hand, should be completely inaccessible (and invisible), if a component is to be used as a black box.

The *specification* of a component is therefore the specification of its *interface*, which must consist of a precise definition of the component's operations and context dependencies, and nothing else.

3 Reasoning about Components

To reason about components and their construction and composition, we will coin a phrase, *a priori reasoning*, which is essential for CBD to achieve its goal of third-party assembly. As its name suggest, *a priori reasoning* takes places *before* the construction takes place, and should therefore provide an *assembly guide* for component composition.

For CBD, *a priori reasoning* would work as follows:

- it requires that it is possible to show *a priori* that the individual components in question are *correct* (wrt their own specifications);

  (This enables us to do *component certification*, see below.)

- it then offers help with reasoning about the *composition* of these components:

  - to guide their composition in order to meet the specification of a larger system;
  - to predict the precise nature of any composite, so that the composite can in turn be used as a unit for further composition.

  (This enables us to do *system prediction*, see below.)

4 Predictable Component Assembly

A priori reasoning addresses an open problem in CBD, viz. *predictable component assembly*. It does so because it enables component certification and system prediction.

Consider Figure 1. Two components A and B each have their own interface and code. If the composition of A and

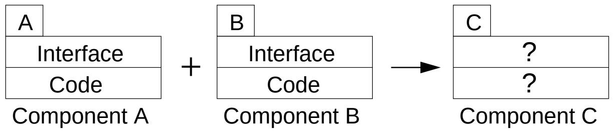

Figure 1. Predicting component assembly.

B is C, can we determine or deduce the interface and code of C from those of A and B? The answer lies in component certification.

4.1 Component Certification

Certification should say what a component does (in terms of its *context dependencies*) and should guarantee that it will do precisely this (for all contexts where its dependencies are satisfied). A certified component, i.e. its interface, should therefore be specified properly, and its code should be verified against its specification. Therefore, when using a certified component, we need only follow its interface. In contrast, we cannot trust the interface of an uncertified component, since it may not be specified properly and in any case we should not place any confidence in its code.

In the context of *a priori reasoning*, a certified component A is *a priori correct*. This means that:

- A is guaranteed to be correct, i.e. to meet its own specification;
- A will always remain correct even if and when it becomes part of a composite.

This is illustrated by Figure 2, where component A has been

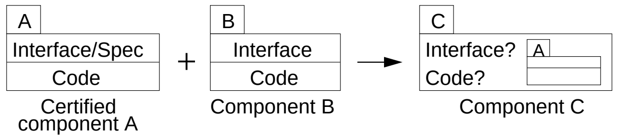

Figure 2. Component certification.

certified, so we know how it will behave in the composite C.

However, we do not know how B will behave in C, since it is not certified. Consequently, we cannot expect to know C's interface and code from those of A and B, i.e. we cannot predict the result of the assembly of A and B.

4.2 System Prediction

For system prediction, obviously we need *all* constituent components to be certified (*a priori correct*). Moreover, for any pair of certified components A and B whose composition yields C:

- *before* putting A and B together, we need to know what C will be;
- and furthermore, we need to be able to certify C.

This is illustrated by Figure 3. The specification of C must

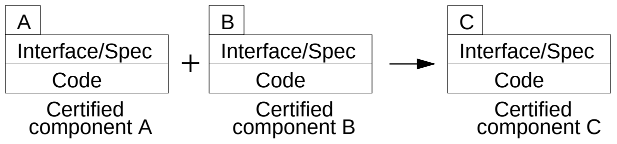

Figure 3. System prediction.

be predictable prior to composition. Moreover, we need to know how to certify C properly, and thus how to use C in subsequent composition. *A priori correctness* is just what we need in order to do system prediction.

5 Modular Specification and Verification

Current approaches to modular (formal) specification and verification, e.g. [9, 14], use *modular reasoning*. This is specification-based reasoning that tries to say before running the software whether it will behave as specified or not (subject to relevant assumptions). This is illustrated in Figure 4. Before a composite module C is deployed, we can

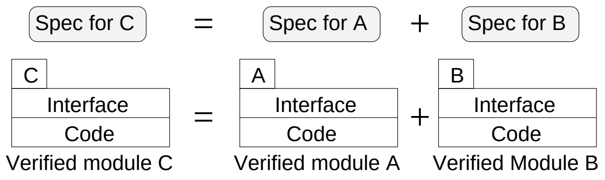

Figure 4. Module composition.

predict whether it will work according to its specification. For example, if component modules, say A and B, are to be used in C, the correctness of C is established based on the specifications of A and B (even before A and B have been implemented). The components A and B are then verified independently. The contexts of A and B are taken in account when using and verifying A and B.

Thus modular reasoning is *a priori* in nature. It predicts correctness, based on specification. This kind of prediction is we believe subtly different from the prediction that we intend to convey in Figure 3, which predicts specification, based on (certified) correctness (we will discuss this in Section 11).

6 Our Approach to Specifying Components

In the rest of this paper, we outline our approach to specifying components, so that we can carry out a priori reasoning about their construction and composition. Our approach differs from current work in modular specification and verification, however, in that we use a novel notion of a priori correctness.

Diagrammatically, our component looks like Figure 5, and in the subsequent sections, we will explain the key ingredients, viz. the *context* and the *interface*, and their specifications.

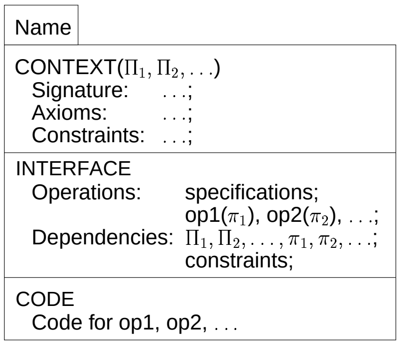
Name
CONTEXT($\Pi_1, \Pi_2, \ldots$)
Signature: ...;
Axioms: ...;
Constraints: ...;
INTERFACE
Operations: specifications;
op1(π_1), op2(π_2), ...;
Dependencies: $\Pi_1, \Pi_2, \ldots, \pi_1, \pi_2, \ldots$;
constraints;
CODE
Code for op1, op2, ...

Figure 5. Ingredients of a component.

We should point out that this is work in progress, so we do not yet have all the answers, so to speak.

7 Context

A component is defined in a *problem domain*, or a *context*. We will represent a context as a full first-order logical theory with an intended (mathematical) model.

7.1 Signature and Axioms

A *context* $\mathcal{C} = \langle \Sigma, X \rangle$ is composed of a *signature* Σ (containing *sort* symbols, *function* declarations and *relation* declarations) and a finite or recursive set X of Σ-axioms. A context axiomatises a problem domain and thus enables us to reason about it. More specifically, a context contains the *abstract data types* (ADTs) and all the concepts that are needed to build a model of the application at hand. A context is thus a (first-order) theory with an intended model.

We distinguish between *closed* and *open* (or *parametric*) contexts. A context $\mathcal{C} = \langle \Sigma, X \rangle$ is *closed* if its signature Σ does not contain any parameters. In this case, $\mathcal{C}$'s axioms X have one fixed model. By contrast, a context $\mathcal{C} = \langle \Sigma, X \rangle$ is *open* if its signature Σ contains parameters. In this case, $\mathcal{C}$'s axioms X have many potential models, depending on the parameters in the signature Σ.

Example 7.1 A simple example of a closed context is first-order arithmetic $\mathcal{NAT} = \langle \Sigma_{PA}, PA \rangle$. Σ_{PA} contains the unary function s (successor) and the binary functions $+$ (sum) and $*$ (product). PA contains the usual Peano's axioms for $s, +, *$ (and all the instances of the first-order induction schema).

CONTEXT $\mathcal{NAT}$;
SIGNATURE:
  Sorts: N;
  Functions: $0 \;:\; [\,] \to N$;
    $s \;:\; [N] \to N$;
    $+, * \;:\; [N, N] \to N$;
AXIOMS: $\{s\} \;:\; \forall x : N \,.\, \neg s(x) = 0$;
    $\forall x, y : N \,.\, s(x) = s(y) \to x = y$;
  $\{+\} \;:\; \forall x : N \,.\, x + 0 = x$;
    $\forall x, y : N \,.\, x + s(y) = s(x + y)$;
  $\{*\} \;:\; \forall x : N \,.\, x * 0 = 0$;
    $\forall x, y : N \,.\, x * s(y) = (x * y) + x$.

The standard structure of natural numbers is the intended model of $\mathcal{NAT}$.

Example 7.2 A simple example of an open context is the following, which axiomatises lists with generic elements X and a generic total ordering $\lhd$ on X.

CONTEXT $\mathcal{LIST}(X, \lhd : [X, X])$;
IMPORT: $\mathcal{NAT}$;
SIGNATURE:
  Sorts : X, L;
  Functions: $nil \;:\; [\,] \to L$;
    $| \;:\; [X, L] \to L$;
    $nocc \;:\; [X, L] \to N$;
  Relations: $pos \;:\; [X, N, L]$;

AXIOMS:
$\{nil, |\} : \forall x, y, z : X,\ \forall j, k, l : L\,.$
  $(\neg nil = x|j \wedge (y|k = z|l \to y = z \wedge k = l))$;
$\{nocc\} : \forall x : X \,.\, nocc(x, nil) = 0$;
  $\forall x, y : X,\ \forall l : L\,.$
  $x = y \to nocc(x, y.l) = nocc(x, l) + 1$;
  $\forall x, y : X,\ \forall l : L\,.$
  $\neg x = y \to nocc(x, y.l) = nocc(x, l)$;
$\{pos\} : \forall x : X,\ \forall l : L\,.$
  $(pos(x, 0, l) \leftrightarrow \exists y : X,\ j : L \,.\, l = y|j \wedge x = y)$;
  $\forall x : X,\ \forall l : L\,.$
  $(pos(x, s(i), l) \leftrightarrow \exists y : X,\ j : L\,.$
  $l = y|j \wedge pos(x, i, j))$.

The context (ADT) $\mathcal{NAT}$ is imported, together with its signature Σ_{PA} and axioms PA.

nil and $|$ are the *constructors* for the sort L of lists of elements of sort X. (For an element x and a list y, $x|y$ stands for the list with head x and tail y.) Their axioms are the list *constructor axioms* (plus structural induction).

$pos(x, i, l)$ means that the element x occurs at position i in the list l, where positions start from 0.

$nocc(x, l)$ is the number of occurrences of the element x in the list l.

7.2 Constraints

In an open context, some of the parameters in the signature may not be instantiated just anyhow. In fact their instantiation must be subject to strictly defined constraints.

Example 7.3 In the context $\mathcal{LIST}(X, \lhd : [X, X])$, in order to ensure that $\lhd$ is a total ordering, we have to add the following *constraints*:

CONTEXT $\mathcal{LIST}(X, \lhd : [X, X])$;
IMPORT: $\mathcal{NAT}$;
SIGNATURE:
Sorts : X, L;
Functions: ...
Relations: ...
AXIOMS: ...
CONSTRAINTS: $\forall x, y, z : X \,.\, (x \lhd y \wedge y \lhd x) \leftrightarrow x = y$;
$\forall x, y, z : X \,.\, (x \lhd y \wedge y \lhd z) \rightarrow x \lhd z$;
$\forall x, y, z : X \,.\, x \lhd y \vee y \lhd x$.

The purpose of constraints is to filter out illegal parameters of the context: only parameters that satisfy the constraints are allowed. For example, if in the context $\mathcal{LIST}($ $X, \lhd : [X, X])$, we want to substitute X by the sort N of natural numbers, and the ordering $\lhd$ by $\leq$ on N, then we can express this as a *closure* (or *instance*):

CLOSURE $\mathcal{NATLIST}$ OF $\mathcal{LIST}(X, \lhd : [X, X])$;
CLOSE: X BY N;
$\lhd$ BY $\forall x, y : N \,.\, (x \lhd y \leftrightarrow x \leq y)$.

This closure of the context $\mathcal{LIST}(X, \lhd : [X, X])$ satisfy the constraints of the context since $\leq$ is a total ordering on N.

In this example, we have closed X and $\lhd$ within $\mathcal{LIST}$ itself, for simplicity. In general, of course, they could also be closed within another context $\mathcal{C}$, after importing $\mathcal{LIST}$ into $\mathcal{C}$.

Obviously constraints define context dependencies.

8 Interface

The interface of a component is defined in the context of the component. The interface is the only part of the component that is visible to the users, and it should provide all the information that the users need in order to deploy the component. Since the interface is defined within the context, the latter should be regarded as part of the former. As we already made clear, the interface should contain specifications for the operations, and the context dependencies, of the component.

8.1 Operations

In the interface, operations are represented by their specifications. In a context $\langle \Sigma, X \rangle$, a specification of a new (relation) symbol r is a set of axioms that define r in terms of the symbols of the signature Σ. For example, suppose in $\mathcal{LIST}$ we have operations for sorting, such as *insertion sort* and *bubble sort*. The specification for these two operations are as follows:[1]

$$\forall l : L \,.\, ord(l) \leftrightarrow$$
$$\forall i : N,\ \forall x, y : X \,.\, ((pos(x, i, l) \wedge pos(y, s(i), l)) \rightarrow x \lhd y)$$
$$\forall j, k : L \,.\, perm(j, k) \leftrightarrow \forall x : X \,.\, nocc(x, j) = nocc(x, k)$$
$$\forall j, k : L \,.\, sort(j, k) \leftrightarrow perm(j, k) \wedge ord(k)$$
$$\forall j, k, l : L \,.\, ord(j) \wedge ord(k) \rightarrow$$
$$(merge(j, k, l) \leftrightarrow ord(l) \wedge perm(j||k, l)).$$

We represent operations as logic programs. For example, the operations *insertion sort* and *bubble sort* are represented by the following logic programs:

Operation: insertionSort($merge$)		
$sort([\,], [\,])$	$\leftarrow$	
$sort(x.j, l)$	$\leftarrow$	$sort(j, k), merge([x], k, l)$

Operation: bubbleSort($\lhd$)		
$sort([\,], [\,])$	$\leftarrow$	
$sort(x.j, y.l)$	$\leftarrow$	$part(x.j, [y], k), sort(k, l)$
$part([\,], [\,], [\,])$	$\leftarrow$	
$part([x], [x], [\,])$	$\leftarrow$	
$part(x.j, [x], y.l)$	$\leftarrow$	$x \lhd y, part(j, [y], l)$
$part(x.j, [y], x.l)$	$\leftarrow$	$y \lhd x, part(j, [y], l)$

The operation insertionSort computes the relation $sort$ (as specified by the specification given above) in terms of the relation $merge$ (also as specified above). It therefore needs a program for $merge$ in order to complete the sorting operation. As a result insertionSort has $merge$ as a parameter, hence we write insertionSort($merge$). In any context that is a closure (instance) of $\mathcal{LIST}$, insertionSort will need a program for $merge$.

[1]For lists j and k, $j||k$ stands for their concatenation.

Thus parameters to operations also define context dependencies.

By contrast, the operation bubbleSort has only the parameter $\lhd$, which is the parameter of the context. So bubbleSort will work for any context in which $\lhd$ is instantiated (closed) by any total ordering.

8.2 Context Dependencies

These consist of the (global) parameters in the signature of the component, the (local) parameters of the operations, together with the constraints in the context.

So now we can define the context dependencies completely in a component.

9 Code

The code should be inaccessible (invisible) to the user. It is usually binary. However, if we allow parameters in the operations, then the code has to be source code, which has to be instantiated before execution.

If the source code is available, then the user or the developer can also verify its correctness with respect to the specifications in the context.

10 A New Notion of A Priori Correctness

In our work the basis for a priori reasoning is a new notion of a priori correctness. So having laid out the specification of a component, we now turn to our definition of a priori correctness of a component. Specifically, we consider a notion of a priori correctness of the operations in a component, that we call *steadfastness*.

10.1 Steadfastness

A *steadfast* operation (program) Op is one that is correct (wrt to its specification) in each intended model of the context $\mathcal{C}$ of the component. Since the (reducts of the) intended models of its specialisations and instances are intended models of $\mathcal{C}$, a steadfast program Op is correct, and hence correctly reusable, in all specialisations and instances of $\mathcal{C}$.

A formalisation of steadfastness is given in [8], with both a model-theoretic, hence *declarative*, characterisation and a proof-theoretic treatment of steadfastness. Here we give a simple example (based on an example in [8]) to illustrate the intuition behind steadfastness.

Example 10.1 Consider the following component:

Iterate	
CONTEXT $\mathcal{ITER}(D, \circ, e)$	
INTERFACE	
Operations:	$S_{iterate}, S_{unit}, S_{op}$;
	iterate$(unit, op)$;
Dependencies:	$D, \circ, e, unit, op$;
CODE	
Code for iterate	

Figure 6. The Iterate component.

where the open context $\mathcal{ITER}(D, \circ, e)$ is defined as follows:

CONTEXT $\mathcal{ITER}(D, \circ, e)$;
IMPORT: $\mathcal{NAT}$;
SIGNATURE:
Sorts: D;
Functions: $e \ : \ [\,] \to D$;
$\circ \ : \ [D, D] \to D$;
$\times \ : \ [D, N] \to D$;
AXIOMS: $\forall x : D\,.\ \times(x, 0) = e$;
$\forall x : D,\ \forall n : N\,.\ \times(x, s(n)) = \times(x, n) \circ x$.

where $\mathcal{NAT}$ is the closed context for first-order Peano arithmetic defined in Example 7.1.

In the open context $\mathcal{ITER}(D, \circ, e)$:

(i) D is a (generic) domain, with a binary operation $\circ$ and a distinguished element e (see the first axiom);

(ii) the usual structure of natural numbers is imported;

(iii) the function symbol $\times$ represents the iteration operation $\times(a, n) = e \circ a \underbrace{\circ \cdots \circ}_{(n \text{ times})} a$ (see the second axiom).

We can use the Iterate component to iterate n times the binary operation $\circ$ on some (generic) domain D.

Suppose in Iterate, or more precisely its context $\mathcal{ITER}$, we specify the $iterate$ operation by the following relation:

$$S_{iterate}: \qquad iterate(a, n, z) \leftrightarrow z = \times(a, n) \qquad (1)$$

The predicate $iterate(x, n, z)$ means that z is the result of applying the iteration operation $\times$ to (a, n), i.e. $z = \times(a, n) = e \circ a \underbrace{\circ \cdots \circ}_{(n \text{ times})} a$.

This specification of $iterate$ can be implemented by the operation iterate$(unit, op)$ defined by the following logic program:

Operation: iterate$(unit, op)$		
$iterate(a, 0, v)$	$\leftarrow$	$unit(v)$
$iterate(a, s(n), v)$	$\leftarrow$	$iterate(a, n, w), op(w, a, v)$

where s is the successor function for natural numbers, and the relations $unit$ and op are specified in $\mathcal{ITER}$ by the specifications:

$$\begin{array}{llcl} S_{unit} : & unit(u) & \leftrightarrow & u = e \\ S_{op} : & op(x,y,z) & \leftrightarrow & z = x \circ y \end{array} \quad (2)$$

The predicate $unit(u)$ means u is the distinguished element e, and $op(x,y,z)$ means that z is the result of applying the operation $\circ$ just once to x and y. Therefore in the program for iterate, if $unit(v)$, i.e. v is just e, then $iterate(a,0,v)$ computes $\times(a,0) = v = e$. Otherwise, if $iterate(a,n,w)$, i.e. $w = \times(a,n)$, and $op(w,a,v)$, i.e. $v = w \circ a$, then $iterate(a,s(n),v)$ computes $\times(a,s(n)) = v = w \circ a = \times(a,n) \circ a = e \circ a \underbrace{\circ \cdots \circ}_{(n+1 \text{ times})} a$.

The operation iterate($unit$, op) is defined in terms of the parameters $unit$ and op. If we can assume that operations for $unit$ and op are a priori correct, i.e. they are correct wrt their specifications (2) in *any* interpretation of $\mathcal{ITER}$, then we can prove that the operation iterate($unit$, op) is *steadfast*, i.e. it is always correct wrt (1) (and (2)).

For example, suppose we have a component Naturals as shown in Figure 7, in which the context is $\mathcal{NAT}$, and the

Naturals
CONTEXT $\mathcal{NAT}$
INTERFACE Operations: S_{unit}, S_{op}; unit, op;
CODE Code for unit, op

Figure 7. The Naturals component.

operations unit and op are specified as follows:

$$\begin{array}{llcl} S_{unit} : & unit(u) & \leftrightarrow & u = 0 \\ S_{op} : & op(x,y,z) & \leftrightarrow & z = x + y \end{array} \quad (3)$$

(i.e. $unit(u)$ means u is 0, and $op(x,y,z)$ means $z = x+y$) and defined as follows:

Operation:- unit
$unit(0).$

Operation:- op
$op(x,y,z) \leftarrow z = x + y$

Then in Naturals, unit and op are (trivially) a priori correct wrt to their specifications (3), and if we compose the components Iterate and Naturals, the operation iterate in the composite Iterate+Naturals will be fully instantiated (and therefore executable), and more importantly it will be correct wrt its specification (1) (and (2)).

The composition here is of course just the simple closure operation on Iterate, but it is sufficient to illustrate the idea of steadfastness. In this closure of Iterate:

(i) D is the set of natural numbers;

(ii) $\circ$ is $+$;

(iii) e is 0;

(iv) $\times(a,n) = 0 + a + \cdots + a = na$.

Consequently, the specification $S_{iterate}$ (1) specialises to

$$iterate(x,n,z) \leftrightarrow z = na$$

and similarly S_{unit} (in (2)) specialises to

$$unit(u) \leftrightarrow u = 0$$

(in (3)), and S_{op} (in (2)) to

$$op(x,y,z) \leftrightarrow z = x + y$$

(in (3)). Since, the operations unit and op are correct with respect to their (specialised) specifications (3), the operation (iterate($unit$, op)$\cup$ unit $\cup$ op) will compute na, and is correct wrt its (specialised) specification in Iterate+Naturals.

To illustrate the correct reusability of the iterate operation in Iterate, suppose now we have a component Integers

Integers
CONTEXT $\mathcal{INT}$
INTERFACE Operations: S_{unit}, S_{op}; unit, op;
CODE Code for unit, op

Figure 8. The Integers component.

as shown in Figure 8, where the operations unit and op are specified by:

$$\begin{array}{llcl} S_{unit} : & unit(u) & \leftrightarrow & u = 0 \\ S_{op} : & op(x,y,z) & \leftrightarrow & z = x - y \end{array} \quad (4)$$

and defined by:

Operation:- unit
$unit(0).$

Operation:- op
$op(x,y,z) \leftarrow z = x - y$

Obviously the operations unit and op in Integers are a priori correct wrt their specifications (4). We can compose Iterate and Integers by a closure operation on Iterate, and get a correct iterate operation in the composite Iterate+Integers.

In Iterate+Integers:

(i) D is the set of integers;

(ii) $\circ$ is $-$;

(iii) e is 0;

(iv) $\times(a,n) = 0 - a - \cdots - a = -na$,

and the specification $S_{iterate}$ ((1) in Iterate) specialises to

$$iterate(x,n,z) \leftrightarrow z = -na$$

S_{unit} (in (2)) specialises to

$$unit(u) \leftrightarrow u = 0$$

(in (4)), and S_{op} (in (2)) to

$$op(x,y,z) \leftrightarrow z = x - y$$

(in (4)). Since unit and op are correct wrt their specifications (4), the operation (iterate($unit$, op)$\cup$unit$\cup$op) computes $-na$ for an integer a, and is correct wrt its (specialised) specification in Iterate+Integers.

The iterate operation is thus a priori correct in Iterate and we say it is *steadfast*. It can be correctly reused in any composite with operations for unit and op as long as these operations are in turn steadfast.

The component Iterate has no constraints in its context dependencies. To further illustrate the notion of steadfastness, we now consider a component whose context dependencies include constraints.

Example 10.2 Consider the component Iterate* (Figure 9) obtained from Iterate (Figure 6) by adding the following

Iterate*	
CONTEXT $\mathcal{ITER}(D,\circ,e)$	
INTERFACE	
Operations:	$S_{iterate}, S_{unit}, S_{op}$; iterate*($unit$, op);
Dependencies:	$D, \circ, e, unit, op$; constraints;
CODE	
Code for iterate*	

Figure 9. The Iterate component.

constraints to its context dependencies:

$$\begin{array}{l} \forall x : D \,.\, e \circ x = x \\ \forall x,y,z : D \,.\, x \circ (y \circ z) = (x \circ y) \circ z \end{array} \tag{5}$$

(these constraints stipulate that $\circ$ should be associative) and by replacing the iterate operation in Iterate by the following operation iterate*:

Operation: iterate*($unit$, op)		
$iterate(a,0,v)$	$\leftarrow$	$unit(v)$
$iterate(a,n,v)$	$\leftarrow$	$m+m=n, iterate(a,m,w), op(w,w,v)$
$iterate(a,n,v)$	$\leftarrow$	$m+s(m)=n, iterate(a,m,w), op(w,w,z), op(z,a,v)$

The operation iterate* has the same specification $S_{iterate}$ (1) as iterate in Iterate, but it computes the relation $iterate$ more efficiently than iterate: the number of recursive calls is linear in iterate, whereas it is logarithmic in iterate*. However, iterate* would *not* be steadfast in Iterate. For example, if we were to use iterate* in place of iterate in Iterate, then iterate* would be correct wrt (1) and (2) in Iterate+Naturals, but it would not be correct wrt (1) and (2) in Iterate+Integers, where, for instance, for

$$iterate(a, s(s(s(s(0)))), v)$$

iterate* would compute 0 instead of the correct answer $-4a$. Thus despite the a priori correctness of unit and op in both Naturals and Integers, iterate* would not be correct in both Iterate+Naturals and Interate+Integers. Therefore iterate* would *not* be steadfast in Iterate.

The reason for this is that in Iterate*, the constraints (5) require that the parameters e and $\circ$ of the context satisfy the unit and associativity axioms. These imply that

$$\begin{cases} \times(a,n) = \times(a, n \div 2) \circ \times(a, n \div 2) \circ a & \text{if } n \text{ is odd} \\ \times(a,n) = \times(a, n \div 2) \circ \times(a, n \div 2) & \text{if } n \text{ is even} \end{cases}$$

which means that whenever $\circ$ is associative, $\times$ can be computed in logarithmic time. So, if we were to use iterate* in place of iterate in Iterate, then iterate* would be correct in Iterate+Naturals because here (D is the set of natural numbers) e is 0, $\circ$ is $+$, and so they actually satisfy the constraints (5) anyway, even though these constraints are not present in Iterate. On the other hand, iterate* would not be correct in Iterate+Integers because here (D is the set of integers) e is 0, $\circ$ is $-$, and since $-$ is not associative, they do not satisfy (5).

However, we can prove that iterate* is steadfast in Iterate*, again assuming a priori correctness of operations for unit and op defined in some other component. It will be correct in any composite Iterate*+C as long as C satisfies the constraints (5) in the context dependencies of Iterate*. For example, as can be seen from the above discussion, iterate* will be correct in Iterate*+Naturals since $+$ is associative.

For something completely different, suppose Matrices is a component with a context of m-dimensional square matrices. Then in the composite Iterate*+Matrices, D is the set of m-dimensional square matrices, e is the m-dimensional identity matrix, and since matrix multiplication $\times$ is associative, iterate* will be correct, where op computes matrix products.

11 Discussion

Since a steadfast program is correct, and hence correctly reusable, in all specialisations and instances of its context, a component with steadfast operations, which we will call a *steadfast component*, when composed with another steadfast component will also be steadfast. In other words, steadfastness is not only *compositional*, but is also preserved through *inheritance* hierarchies.

Consequently, in the context of system prediction (as shown in Figure 3) when composing steadfast components, not only can we be sure that the composite will be steadfast, but we can also predict the specification of the composite. This is illustrated in Figure 10.

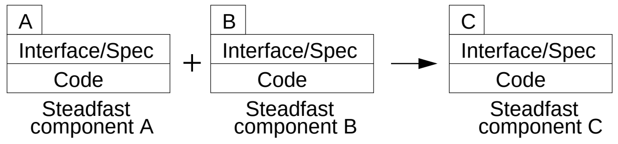

Figure 10. Composing steadfast components.

In the context of modular specification and verification (as shown in Figure 4), steadfast modules can be verified and the specification of the composite can be predicted, prior to composition. This is illustrated in Figure 11. We understand that current approaches to modular reasoning need to know the specification of the composite before predicting if the composite will work according to its specification. If this is the case (as shown in Figure 4), then steadfastness offers the advantage of being able to predict the specification of the composite prior to composition. Thus, with steadfast modules, we can do system prediction as shown in Figure 10.

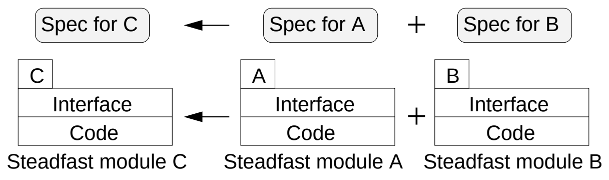

Figure 11. Composing steadfast modules.

12. Conclusion

For lack of space, we have presented the intuition behind steadfastness by means of simple examples. We hope this does not detract from its presentation.A full account of steadfastness can be found in [8]. Steadfastness is defined in terms of model-theoretic semantics. It is thus declarative in nature. We believe that declarative semantics in general will be important for lifting the level of abstraction.

Our approach to specifying components is very generic. The component may be just a class or ADT. It may be a module, in particular what Meyer [10] calls an *abstracted module*, which is the basic unit of reuse in the CBD methodology RESOLVE [14]. It may be an object model [2] as in OMT [11] or UML [12]. It may yet be an OOD framework, i.e. a group of interacting objects [6], such as frameworks in the CBD methodology *Catalysis* [3, 5]. It could even be a design pattern or schema [4].

We believe that our approach to component specification can enable predictable component assembly, which is currently an open problem in CBD. In addition, we believe it can provide a hybrid, spiral approach to CBD [7] that is both top-down and bottom-up for CBD, as illustrated in Figure 12. First a library of steadfast components has to be built. The nature of steadfastness, coupled with the use of *a priori reasoning*, then allows these components to be composed into larger systems in either a top-down (following the traditional *waterfall model* or the *software architecture* approach [13, 1]), or bottom-up manner, or indeed a combination of both.

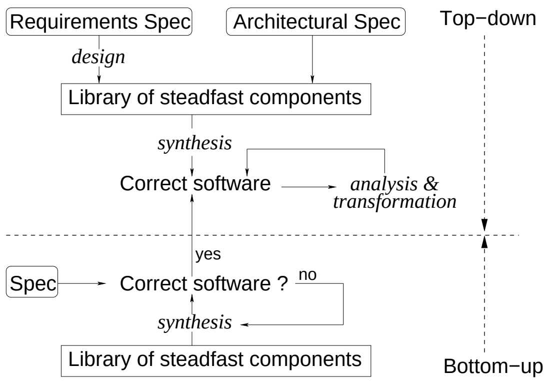

Figure 12. A spiral model for CBD.

Bottom-up development in particular is more in keeping with the spirit of CBD. Composition of steadfast components can show the specification of the composite, and therefore the specification of any software constructed can be compared with the initial specification for the whole system. Guidance as to which components to 'pick and mix' can also be provided by component specifications.

Acknowledgements

We are grateful to Murali Sitaraman and the reviewers for their helpful comments in general, and for correcting our misunderstanding of modular specification and verification in particular.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[2] R. Bourdeau and B. H. Cheng. A formal semantics for object model diagrams. *IEEE Trans. Soft. Eng.*, 21(10):799–821, 1995.

[3] D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.

[4] P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An abstract formalisation of correct schemas for program synthesis. *Journal of Symbolic Computation*, 30(1):93–127, July 2000.

[5] J. Küster Filipe, K.-K. Lau, M. Ornaghi, K. Taguchi, A. Wills, and H. Yatsu. Formal specification of Catalysis frameworks. In J. Dong, J. He, and M. Purvis, editors, *Proc. 7th Asia-Pacific Software Engineering Conference*, pages 180–187. IEEE Computer Society Press, 2000.

[6] J. Küster Filipe, K.-K. Lau, M. Ornaghi, and H. Yatsu. On dynamic aspects of OOD frameworks in component-based software development in computational logic. In A. Bossi, editor, *Proc. LOPSTR 99, Lecture Notes in Computer Science*, volume 1817, pages 43–62. Springer-Verlag, 2000.

[7] K.-K. Lau. Component certification and system prediction: Is there a role for formality? In I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau, editors, *Proceedings of the Fourth ICSE Workshop on Component-based Software Engineering*, pages 80–83. IEEE Computer Society Press, 2001.

[8] K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, 38(3):259–294, March 1999.

[9] G. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, pages 72–80, July 1991.

[10] B. Meyer. *Object-oriented Software Construction.* Prentice-Hall, second edition, 1997.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Sorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[12] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[13] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[14] M. Sitaraman and B. Weide, editors. *Component-based software using RESOLVE*. Special feature, ACM Sigsoft Software Engineering Notes 19(4): 21-65, October 1994.