

Reasoning About Concurrent Objects

Heinz W. Schmidt and Jian Chen

Department of Software Development, Monash University, Melbourne,
Caulfield VIC 3145, Australia

{Heinz.Schmidt,Jian.Chen}@fcit.monash.edu.au

Abstract

Embedded specifications in object-oriented (OO) languages such as Eiffel and Sather are based on a rigorous approach towards validation, compatibility and reusability of sequential programs. The underlying method of “design-by-contract” is based on Hoare logic for which concurrency extensions exist. However concurrent OO languages are still in their infancy. They have inherently imperative facets, such as object identity, sharing, and synchronisation, which cannot be ignored in the semantics. Any marriage of objects and concurrency requires a trade-off in a space of intertwined qualities.

This paper summarises our work on a type system, calculus and an operational model for concurrent objects in a minimal extension of the Eiffel and Sather languages (cSather). We omit concurrency control constructs and instead use assertions as synchronisation constraints for asynchronous functions. We show that this provides a framework in which subtyping and concurrency can coexist.

1 Introduction

Concurrent objects (CO) are a very active field of research in computer science given the growth of distributed and parallel processing. There are many proposals for a marriage of distributed computing and object technology[13, 12]. Most of them are unsatisfactory at present. One of the oldest school is the ACTOR family of languages. Actors, however, lack intra-object concurrency and do not give programmers control of the granularity of concurrency. They rely on advanced compiler technology to achieve moderate efficiency. Declarative (functional or logical) object languages mostly fall into two schools: parallelisation of sequential calculi and imperative concurrency extensions. Only limited efficiency can be achieved by pure parallelisation. For the latter the interference of con-

trol with the underlying pure calculi often poses fundamental problems[4, 20]. All of the practically used CO languages are imperative, partly dropping objects into existing concurrent languages (e.g. Modula3) or adding concurrency to objects (e.g. Argus, CEiffel, pSather). Some take a library approach adding to a core sequential language (e.g. various C++ extensions) and ignore semantic issues altogether.

Many of the new languages are not yet viable for industrial use. Few consider the problem of composing concurrent objects without recompilation, especially with a view of proven guarantees of service. However increasingly safety or mission critical systems are being distributed, compliance with international quality standards are becoming mandatory. CO therefore necessitate trade-offs particularly between provability (of global properties), local control (of encapsulated objects) and efficiency.

We are aiming towards a minimal and practical approach to provability for CO based on Eiffel[8] and Sather [10, 11, 14] in an extension of Sather called *cSather* (concurrent Sather). Hoare triples [18] are part of the visible interface and become immediately useful in systematic validation, documentation and reuse, whether supported by tools or as part of an external quality management process. A methodology called “*design-by-contract*” [7] is a genuinely incremental approach to class reuse and system stability [15].

The so-called inheritance anomaly[5] poses a difficulty specific to CO. It has led some to abandon inheritance altogether. We believe this is pouring out the baby with the bath water. Ferenczi [3] proposes a return to Brinch Hansen’s nested critical sections, which work with subtyping, but are known to be not efficiently implementable in general. The present paper extends an existing semantics[16, 17] of design-by-contract to concurrency without imperative concurrency constructs. Instead, interface definitions expose object distribution and interaction constraints, which then induce an object-oriented notion of concurrent executions of the seemingly sequential codes.

```

class STACK(T) is
  -- Implementation of STACK by flexible arrays.
  stack: ARRAY(T) := ARRAY(T).new(size := initsize);
  ssize: INT := 0;
  constant initsize = 5;
  invariant safe = (0 <= ssize and ssize <= stack.size);

  create: SAME is res := SAME.new end;

  push(x : T) is
    h: ARRAY(T);
    if stack.size = ssize then -- extend stack
      h := ARRAY(T).new(size := 2 * ssize);
      ... -- copy all elements of stack into h
      stack := h;
    end;
    stack[ssize] := x;
    ssize := ssize + 1;
  end; -- push

  pop pre not empty is ssize := ssize - 1 end;

  top : T pre not empty is res := stack[ssize - 1] end;

  empty : BOOL is res := (ssize = 0) end;

end; -- STACK

```

Figure 1: A Sather Class

2 Types and Interfaces

Typing introduces constraints on object spaces. These include requirements regarding the existence of attributes or functions or the validity of formulas such as invariants for the current object “*Self*”. Our type system on which such formulas are built, is called TOF (Types for Objects and Functions). It supports sub-type polymorphism and passing named polymorphic functions as parameter. TOF has modeled and influenced the type system of the Sather language [19].

Sather distinguishes subtyping, i.e., conformance between types of objects, from subclassing, i.e., inheritance or partial reuse of code. Designers can insert new types below, above or in between existing types. With this ability intermediate classes can be introduced when needed and OO designs avoid “over-classification”. TOF is concerned with the subtype hierarchy only, not with code inheritance.

2.1 Type Expressions

Product and function types are built from simple types, introduced with subtyping below. Simple types model both referenced objects and value types.

```

class MAIN is
  main is
    i : INT := 0;
    s: STACK(INT) := STACK(INT).create;
    while i <= 10 do
      s.push(i);
      i := i + 1;
    end;
    while not s.empty do
      OUT.s.top;
      s.pop;
    end;
  end; -- main
end; -- MAIN

```

Figure 2: A Sather Program

Definition 2.1 (Simple types) *The pair $\langle S, \prec_S \rangle$ is a set of simple types if S is a finite set (of names) and \prec_S is a binary, irreflexive, acyclic relation over S . We often write $\langle S, \prec \rangle$ as simply S . \preceq denotes the partial order (of subtyping) induced by \prec .*

The semantics of simple types can be based on an order-sorted algebra \mathbf{D} in which each sort is interpreted as a set of values and \preceq is interpreted as the subset relation of *value domains*.

Figures 1 and 2 show two Sather classes. *res* in function bodies is the returned result (e.g. *top* in Figure 1). We assume the basic types *INT*, *BOOL* and *ARRAY(T)* (one-dimensional flexible arrays) are given with the usual semantics.

Definition 2.2 (Type expressions) *Let S be a set of simple types. The set of type expressions, written S^+ , and the set of type expressions with void, written S^* are defined inductively:*

- $\mathbf{1} \in S^*$ (called *void type*) and $S \subseteq S^+ \subseteq S^*$.
- If $\alpha_1, \dots, \alpha_n \in S^+$ then $\alpha_1 \times \dots \times \alpha_n \in S^+$ (*product type*).
- If $\alpha, \gamma \in S^*$ and $\beta \in S^+$, then $[\alpha \rightarrow \beta] \in S^+$ and $[\alpha \rightarrow \gamma] \in S^*$ (*monomorphism type*).
- If $\alpha \in S$ and $\beta \in S^+, \gamma \in S^*$, then $[\alpha \rightsquigarrow \beta] \in S^+$ and $[\alpha \rightsquigarrow \gamma] \in S^*$ (*polymorphism type*).

In function types $[\alpha \rightarrow \beta]$ and $[\alpha \rightsquigarrow \beta]$, α is called the *arity type* and β the *result type* respectively. Note that the polymorphism arity types are restricted to simple types only.

Relations \prec and \preceq extend to type expressions inductively by the following subtyping rules (excluding the reflexivity and transitivity of \preceq).

$$\frac{\alpha_1 \preceq \beta_1, \dots, \alpha_n \preceq \beta_n}{\alpha_1 \times \dots \times \alpha_n \preceq \beta_1 \times \dots \times \beta_n}$$

$$\frac{\alpha \preceq \beta, \gamma \preceq \delta}{[\beta \rightarrow \gamma] \preceq [\alpha \rightarrow \delta]} \quad \frac{\alpha \preceq \beta, \gamma \preceq \delta}{[\beta \rightsquigarrow \gamma] \preceq [\alpha \rightsquigarrow \delta]}$$

2.2 TOF Expressions

We build typeless expressions and subject them to well-formedness and well-typing rules later. To this end, we assume a countable set X ($X = (X_\alpha)_{\alpha \in S^*}$) of *variables* with a distinguished variable *Self* in each of the X_s ($s \in S$). Furthermore there is a finite set F of *function symbols*, where $X \cap F = \emptyset$. F is divided into two disjoint sets F_m and F_p of *monomorphism* and *polymorphism* symbols, respectively. For example, `ARRAY.new` and `STACK.create` in Fig. 1 would be modeled as monomorphism, `push` as polymorphism name. Given X and F , we define *TOF expressions* T as the minimal set satisfying:

- If $x \in X$, then $x \in T$ (variables).
- If $m \in F_m$, then $m \in T$ (monomorphism).
- If $p \in F_p$ and $\alpha \in S$, then $\alpha.p \in T$ (bounded polymorphism).
- If $e_1, \dots, e_n \in T$, then $(e_1, \dots, e_n) \in T$ (tupling).
- If $e \in T$ where $e = (e_1, \dots, e_n)$, then $e.i \in T$ for all $i \in \{1, \dots, n\}$ (projection).
- If $e \in T$ and $m \in F_m$, then $m(e) \in T$ (monomorphism applications).
- If $e \in T$ and $p \in F_p$, then $e.p \in T$ (polymorphism applications).

The set of attributes A is a distinguished subset of F_p . For $a \in A$ we will treat $e.a$ as a reader polymorphism call. Attribute updates are only possible by assignments $a := e$ within an object. External update requests can only be realised by calling a visible polymorphism of that class.

For brevity, we do not define all statements formally, although the languages we consider contain a minimal set of imperative constructs. As an example the assignment statement may suffice, since procedures can be viewed as void functions. In an assignment $a := e$, a is a variable or an attribute and e an expression.

Signatures, i.e., collections of type declarations are now used as the starting point for typing judgement.

Moreover we introduce well-formed rules for signatures to simplify typing.

Definition 2.3 (Signatures) *Let X be a set of variables, F be a set of function symbols and S be a set of types. A signature $\Sigma_{X,F,S}$ (or simply Σ) with respect to X , F and S^* is a finite set of type declarations of the form $a : \alpha$, where $a \in X \cup F$ and $\alpha \in S^*$.*

We will use the following notations.

- $a \in \Sigma$ ($a \notin \Sigma$) means there exists (does not exist) α such that $a : \alpha \in \Sigma$ ($a : \alpha \notin \Sigma$).
- Σ^α is defined as the set $\{a : \beta \mid a : \alpha \rightsquigarrow \beta \in \Sigma\}$ and called the *class signature* of $\alpha \in S$.

Definition 2.4 (Well-formedness) *Let Σ be a signature. We call Σ well-formed if and only if Σ satisfies the following conditions.*

- If $a : \alpha \in \Sigma$, $a : \beta \in \Sigma$ and $\alpha \neq \beta$, then $a \in F_p$ and there exist $\alpha_1, \beta_1 \in S$ and $\alpha_2, \beta_2 \in S^*$ such that $\alpha = [\alpha_1 \rightsquigarrow \alpha_2]$, $\beta = [\beta_1 \rightsquigarrow \beta_2]$ and $\alpha_1 \neq \beta_1$.
- If $\alpha \prec \beta$ and $f : [\beta \rightsquigarrow \beta'] \in \Sigma$ (with $\beta' \in S^*$) then there exists $\alpha' \in S^*$ such that $f : [\alpha \rightsquigarrow \alpha'] \in \Sigma$ and $\alpha' \preceq \beta'$.

The first condition says that overloading is restricted to polymorphisms and polymorphisms dispatch uniquely. The second imposes that polymorphisms are prefix-closed and compatible. It is straightforward to show that well-formedness is decidable.

Well-formedness implies the following (with $\alpha, \beta \in S$ and $\alpha \preceq \beta$):

Lemma 2.5 (Behaviour inclusion) $f \in \Sigma^\beta$ implies $f \in \Sigma^\alpha$. (For short we write $\Sigma^\beta \subseteq \Sigma^\alpha$).

Theorem 2.6 (Signature regularity) Σ is regular, i.e., for all $f : \beta' \in \Sigma^\beta$, there is a unique type α' with $f : \alpha' \in \Sigma^\alpha$ such that $\alpha' \preceq \beta'$.

Corollary 2.7 (Redefinition conformance) For all $f : \alpha_1 \rightarrow \alpha_2 \in \Sigma^\alpha$ and $f : \beta_1 \rightarrow \beta_2 \in \Sigma^\beta$, we have that $\beta_1 \preceq \alpha_1$ and $\alpha_2 \preceq \beta_2$.

2.3 Type Checking

Intuitively, type checking refers to the problem of deriving inductively whether a TOF expression e can be assigned a type $\alpha \in S^*$ using the TOF typing rules below. Formally, we say e is *well-typed* with respect to a well-formed signature Σ if and only if there exists

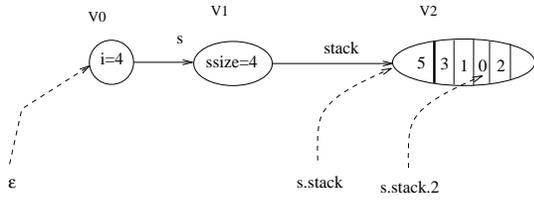


Figure 3: A State of the Sather Program MAIN

$\alpha \in S^*$ such that $\Sigma \vdash e \varepsilon \alpha$, where \vdash is deduction relative to the rules below

$$\text{(Gen)} \quad \frac{}{a \varepsilon \alpha} \quad \frac{}{\beta.f \varepsilon [\beta \rightsquigarrow \gamma]}$$

if $a : \alpha \in \Sigma$, $a \in X \cup F_m$, or $f : [\beta \rightsquigarrow \gamma] \in \Sigma$, $f \in F_p$.

$$\text{(Appl)} \quad \frac{e \varepsilon \alpha, f \varepsilon [\alpha \rightarrow \beta]}{f(e) \varepsilon \beta} \quad \frac{e \varepsilon \alpha, f \varepsilon [\alpha \rightsquigarrow \beta]}{e.f \varepsilon \beta}$$

$$\text{(Tup)} \quad \frac{e_1 \varepsilon \alpha_1, \dots, e_n \varepsilon \alpha_n}{(e_1, \dots, e_n) \varepsilon \alpha_1 \times \dots \times \alpha_n} \quad \frac{e \varepsilon \alpha_1 \times \dots \times \alpha_n}{e.1 \varepsilon \alpha_1 \dots e.n \varepsilon \alpha_n}$$

$$\text{(Coerce)} \quad \frac{e \varepsilon \alpha, \alpha \preceq \beta}{e \varepsilon \beta}$$

The following corresponds to the typing rule for assignment.

$$\text{(Assign)} \quad \frac{e \varepsilon \alpha}{a := e \varepsilon 1} \quad (a \in A, a : \alpha \in \Sigma^t)$$

Note here $a := e \varepsilon 1$ simply says that the assignment statement is well-typed.

We have the following results regarding to type checking but omit their proofs in this paper for brevity.

Theorem 2.8 (Decidability of well-typing) *It is decidable whether a TOF expression is well-typed.*

Theorem 2.9 (Expression regularity) *Let $\alpha, \beta \in S^*$, and $\Sigma \vdash e \varepsilon \alpha$, then there is a unique smallest $\beta \preceq \alpha$ with $\Sigma \vdash e \varepsilon \beta$.*

Theorem 2.10 (Type safety) *Given a signature Σ and a TOF expression e , if $\Sigma \vdash e \varepsilon \alpha$, then the evaluation of e is in the domain of the type α .*

2.4 Interfaces and Stability

So far our types are capturing only signatures. Now we define invariants, pre-conditions and post-conditions as TOF formulas. A TOF formula is a

boolean expression, possibly associated with a label (i.e., a name, for selective redefinition). Invariants I_s (for $s \in S$) are associated to polymorphism arity, pre-conditions $P_{f,s}$ and post-conditions $Q_{f,s}$ to monomorphism arity and result respectively. For brevity, we omit the formal definitions in this paper. Examples can be found in Fig. 1. At the heart of design-by-contract is a *local conformance* constraint that relates the assertions of one function name in a direct subclass $\alpha \prec \beta$ to those in β . We have isolated and formally captured these requirements (weakening Eiffel's requirements):

- $I_\alpha \Rightarrow I_\beta$: invariants strengthen
- $I_\alpha \wedge Q_{f,\alpha} \Rightarrow Q_{f,\beta}$: postconditions strengthen
- $I_\beta \wedge P_{f,\beta} \Rightarrow P_{f,\alpha}$: preconditions weaken

Object-wise local correctness requires that observable states (expression in terms of interface functions) satisfy the invariant of the respective type. *Function-wise local correctness* is defined as follows: the implementation of a function satisfies its post-condition provided its pre-condition is satisfied. These two are expressed by Hoare triples $\{I \wedge P\} o.f(a) \{I \wedge Q\}$, where o is an object, f a function and a the arguments of the call [18]. If these assertions are valid in all possible histories (see below Def. 3.5) for all functions and objects, we speak of *global correctness*.

It turns out that *class-wise local conformance* helps in guaranteeing global correctness.

Theorem 2.11 (Substitutability) *If o is an object of type β ; e_i is of type β_i satisfying $\{I_\beta \wedge P_{f,\beta}\} o.f(e_0, \dots, e_{n-1}) \{I_\beta \wedge Q_{f,\beta}\}$, then the same holds for o substituted by an object of type α such that $\alpha \preceq \beta$, provided o and f are locally correct and the local conformance constraint holds for α and β . Especially it holds for any actual (run-time) value of the formal expression o , and in any future conforming and locally correct extension.*

Theorem 2.12 (Stability) *If all objects and functions are locally correct and all classes locally conforming then the system is globally correct.*

The above theorems imply that we can reason incrementally on the correctness of (concurrent) objects and do not have to have global analysis and proofs if we do not want to: whenever a signature is added, we check for well-formedness and well-typing; whenever invariants, pre- or post-conditions are added to (well-formed) signatures, we check the local conformance; when finally an implementation of functions

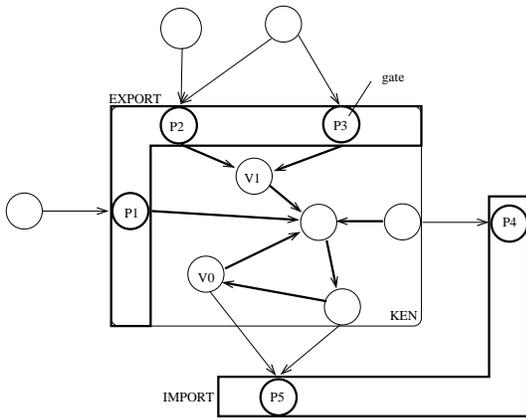


Figure 4: Distribution and colocation of objects

is given, we prove their local correctness relative to the given class interface only. The global correctness then follows from Theorem 2.12. Only if desirable for global interobject optimisation or more detailed analysis need we reason beyond class boundaries.

3 Operational Semantics

In our framework, we do not consider parametric polymorphism (such as in $STACK(T)$). We use a *two-level semantics*. First we transform a program to TOF. Then we model its operational semantics. For these, we start typeless again, representing states as graphs called *object spaces*. OO languages can be defined as *object-based* languages with the addition of classes and inheritance [1]. The language *Self* [2] is object-based in this sense.

CO are then based on three abstractions, *gate*¹ types distinguishing separate objects, synchronisation *constraints* limiting parallel access through gates, and a *semi-synchronous* call semantics giving callees control over caller blocking, and thus allowing redefinition with minimal changes the callers, and a sequential interpretation of the caller code.

Gates control entry to separate regions of colocated objects called Kens². Kens are the smallest units of distribution. A Ken can only be accessed through one of its gates. Computations in different Kens proceed in parallel and can lead to parallel gate calls. In general, different gate calls (to one or more functions) proceed in parallel.

¹Gates are large guarded entrances permitting one or more to enter at the same time.

²A Japanese word for local community or region.

Synchronisation *constraints* restrict such gate calls. *Holding* (of a Ken) delays execution until exclusive access is guaranteed. *Future* conditions (attributes) await definition in the future as a result of ongoing computations. Calls accessing an undefined future attribute block and are queued until the value becomes available (“wait-by-necessity”). Gate calls are *semi-synchronous*. A caller blocks until it receives a reply. The callee returns a result or replies with an as yet undefined future and proceeds computing that.

A Ken is closed under colocation. Its gates are the *only* interface to its environment. Synchronisation constraints are part of the callee class interface not of its implementing code. Kens guarded by synchronisation constraint can thus be used for encapsulating protection domains and access policies.

Since gate calls may proceed computing after returning to the caller, intra-object parallelism is supported within Kens. While a single Ken “houses” threads that share objects and gives rise to “constrained” parallelism, different Kens are disjoint and give rise to “free” parallelism [9]. This distinction is semantically relevant for provability and optimisation.

As CPU speed increases more rapidly than the speed of networks, in modern distributed computing architectures, locality is essential for achieving efficiency. Local and remote objects/threads must be distinct for estimating costs and optimising programs, whether a programmer, compiler or runtime system allocates objects and threads. Threads cross Ken boundaries in a well-defined way semantically. Allocation and migration is possible for entire Kens.

Kens thus generalize Hoare monitors by pattern of related gates rather than one individual class. Several gates can share future condition attributes, which take the role of monitor condition variables. Furthermore these generalised monitors allow multi-threading but mutually exclusive access can be enforced by means of synchronisation constraints.

3.1 Object Spaces

We use labeled graphs to represent the states of a state space and graph transformations to model dynamic changes in that space. We distinguish value and reference semantics clearly starting from a partitioning of attributes A in Val and Ref . Value attributes are part of the graph vertices. References are edges. For an example see Figure 3 which also explains the interpretation of dotted attribute expressions relative to vertex $V0$. $V2$ is an array object.

```

class BUFFER is
  empty: COND(BOOL);
  buffer: ARRAY(INT);
  size, head, tail: INT;

invariant downsafe = (size >= 0);
...
put(item: INT) -- unbounded buffer, keep going
  hold Self
  post initial(size) = size - 1 and item = buffer[tail - 1]
  set empty := false
  is ... end;

get: INT
  on not empty hold Self
  post initial(size) = size + 1 and res = buffer[head - 1]
  set empty := (size = 0)
  is ... end;
end

```

Figure 5: A Buffer Fragment

Definition 3.1 (Object Space) An object is an element of the set $OB = \bigcup_{X \subseteq Val} \mathcal{N} \times \mathcal{S} \times \mathcal{D}^X$. For an object $o = (id = n, type = s, a_0 = v_0, \dots, a_{n-1} = v_{n-1})$ we abbreviate $o.id = n$ (the object identity), $o.type = s$ the type of o , and $o.a_i = v_i$ (the attribute values). A distributed object space is a labeled directed multigraph $G = (V, E, \lambda, Gat)$, such that

- (i) $V \subset OB$,
- (ii) for $v \neq w \in V$ we have: $v.id \neq w.id$ (i.e., objects have a unique identity),
- (iii) $\lambda : E \rightarrow Ref$ is injective on $v \rightarrow$, i.e. $a : V \rightarrow V \uplus \{\perp\}$ is a function. Again, we use the dot notation $v.a$ for $a(v)$.
- (iv) $Gat \subseteq V$. The elements of Gat are called gates.

Let C be the relation vCw iff $v \rightarrow w$ and $w \notin Gat$. The transitive, reflexive and symmetric closure of C is called the gate colocation relation. It partitions the object space. For $v \in V$ we denote the colocation partition of v by $Ken(v)$ and call the set $Ken(v) \cap Gat$ the gates of $Ken(v)$.

Figure 3 shows a state of the program defined in Figures 1 and 2. Vertex $V0$ represents the activation record of function “main” with local variables i and s represented as attributes. Vertex $V2$ represents an array object. Figure 4 illustrates colocation (solid edges) in a distributed object space. Gates (such as $P1$) are represented as solid circles.

Due to the above definitions, G uniquely determines an algebra \mathbf{D}_G extending \mathbf{D} with additional

sorts $D_\alpha := \{o \mid o.type = \alpha\}$ for monotypes, i.e., minimal $\alpha \in S$, and $D_\beta := \bigcup_{\alpha \preceq \beta} D_\alpha$ otherwise. For all attributes $a \in A$ and $a \in \Sigma^\alpha$, there is an associated function a_α such that $a_\alpha(o) = o.a$ for $o \in D_\alpha$. When unambiguous, we equate \mathbf{D}_G with the object space itself. Given a well-formed signature Σ of attribute declarations, we call G a Σ -object space if \mathbf{D}_G is a (order-sorted) Σ -algebra.

The following observations hold true.

Corollary 3.2 (Regularity) For each $\alpha \in S$ and $o \in D_\alpha$ there is a unique smallest $\beta \in S$ such that $o \in D_\beta$. Moreover $\beta = o.type$.

Corollary 3.3 (Well-definedness) $a : \beta \in \Sigma^\alpha$ then $a_\alpha : D_\alpha \rightarrow D_\beta$ and for all $\alpha' \preceq \alpha$ $a_{\alpha'} = a_\alpha|_{D_{\alpha'}}$.

The attribute mappings in \mathbf{D}_G extend to object path expressions $p \in Ref^* \cup Ref^* Val$ in the obvious way. We use dot notation to denote such expressions. For example $v.a.b$ represents $b(a(v))$.

3.2 Multithreading

Since accesses to a separate Ken always go through gates and gates spawn new threads, distribution implies parallelism as an OO form of distributed computation [9]. We model a thread of execution by a selected vertex representing the current function execution frame within the object space. A frame f executes in $Ken(f)$. In expressions, “ Me ” refers to the current frame f . Distinguished attributes of f represent “ $Self$ ” (the current object in V), “ arg_i ” (the i -th parameter in $V \cup D$), “ $Caller$ ” (the caller frame), “ res ” the return value, if any, and finally “ AT ” (the program counter).

Definition 3.4 (Multithreads) A multithread in a distributed object space $G = (V, E, \lambda, Gat)$ is a sequence $t = t_0, \dots, t_{n-1} \in V^*$. t_i is called a thread and is said to execute at statement t_i . At with the current object $t_i.Self$. Moreover, the sequence of vertices $t_i = w_0 \xrightarrow{Caller} \dots \xrightarrow{Caller} w_{m-1} \xrightarrow{Caller} \perp$ is called the stack of t_i , short $Stack(t_i)$. A distributed state space $\mathcal{S}_{D, \Sigma}$ is the set of tuples (V, E, λ, Gat, t) where $G = (V, E, \lambda, Gat)$ is a distributed object space and t is a multithread in G .

Note that local variables and parameters can be values or references. In languages such as Eiffel and Sather, unlike C++, references to value locations cannot be created. This excludes pointers from heap to stack and thus excludes dangling references and the resulting type unsafety as well as “behind-the-back”

```

class BOUNDED_BUFFER subtype BUFFER is
include BUFFER;
full: COND(BOOL);
max: INT;
invariant upsafe = (size <= max);
...
put(item: INT)
  on not full hold Self
  post ...
  set empty := false; full := (size = max)
  is ... end;

get: INT
  on not empty hold Self
  post ...
  set full := false; empty := (size = 0)
  is ... end;
end

```

Figure 6: A Bounded Buffer Fragment

interferences from other threads. The ability to reason about interference freedom is an essential to CO.

3.3 Synchronisation

Mutual exclusion is achieved by holding objects. A **hold** clause in the function prelude lists the objects to be held exclusively for the duration of the function execution. A **hold** function assumes the caller holds *Self*. **hold** parameters are held by the caller.

In our design, we opted that holding an object *o* amounts to holding $Ken(o)$. A strong form of interference freedom of program executions can thus be used for reasoning within Kens, too. The designer/programmer controls the granularity of concurrency and atomicity by the granularity of Kens. Mutual exclusion can be implemented efficiently by associating a lock to each Ken, and grabbing a sequence of locks atomically as part of the function prelude.

While the undefined value \perp represents an error for other expressions, it represents *future*, i.e., delayed evaluation of *condition attributes*. We use a distinguished reference type $COND(T)$ for these. Threads accessing an undefined future block until the attribute is defined and then obtain the value of type *T*. Access to future conditions is restricted to guard and continuation clauses.

The guard clause “**on** $c = i, d$ ” for example tests *c* and *d* for definedness and compares *c*’s value. The continuation clause “**set** $c := v; d$ ” defines *c* and *d* and sets *c*’s value to *v*.

For efficiency, guards are restricted to conjunctions of simple value comparisons, so that blocking threads can be queued on a unique condition requiring update to proceed. When a guard succeeds, all its component

conditions become undefined. Guard success and obtaining hold of objects is an atomic operation. Continuations are defined immediately on exit.

Reference conditions allow several Ken objects to share a future condition and they permit designers to introduce new types of future conditions.

Fig. 5 and 6 illustrate the use of synchronisation constraints in fragments of an unbounded and bounded buffer definition. *BOUNDED_BUFFER* claims to conform to the *BUFFER* specification (*subtype*) and inherits elements of the *BUFFER* implementation (*include*).

Functions *put* and *get* hold *Self* and are thus mutually exclusive. The bounded buffer definition adds a new invariant (axiom) to the specification and uses an additional future condition (*full*).

3.4 Concurrent Interfaces Stability

We have shown in the previous section that design-by-contract leads to stable constructions of OO systems. We wish to extend these notions to concurrent systems. We model holding by an implicit *available* condition for the respective Ken, and can thus restrict synchronisation constraints to Boolean expressions on future conditions.

We abbreviate the corresponding guard and continuation clauses by $G_{f,s}$ and $C_{f,s}$, respectively. Now, we extend the conformance constraints (with $\alpha \prec \beta$) as follows:

- $I_\alpha \wedge P_{f,\alpha} \wedge G_{f,\alpha} \Rightarrow G_{f,\beta}$ guards strengthen
- $I_\alpha \wedge Q_{f,\alpha} \wedge C_{f,\alpha} \Rightarrow C_{f,\beta}$ continuations strengthen

Moreover we extend correctness in terms of Hoare triples to

$$\{I_\alpha \wedge P_{f,\alpha} \wedge G_{f,\alpha}\} o.f(a) \{I_\alpha \wedge Q_{f,\alpha} \wedge C_{f,\alpha}\}$$

The substitutability and stability theorems hold with these modifications for concurrent histories, under the assumption that preconditions, once established by a caller, are established if and when, eventually, the callee proceeds. This is a proof obligation and can often be guaranteed by sufficient restrictions on the form of preconditions. Correctness is then interpreted as the validity of postcondition and continuation clause immediately on exit.

At this point, a more intuitive explanation of the conformance seems in order. A correct β object (I_β holds) is expected to delay execution of a correct call ($P_{f,\beta}$ holds) while $\neg G_{f,\beta}$. By conformance these imply $\neg G_{f,\alpha}$ and therefore the delay of the subtype function

(of the α -object). Thus the subtype function satisfies the β -constraint. It is interesting to note that for substitutability, the implication on exclusion constraints goes from supertype to subtype like that for preconditions. Of course this is equivalent to strengthening the guard. When the subtype guard is eventually satisfied, by the assumption of the theorem the precondition holds. By conformance also the supertype guard would hold and the firing of the supertype function would therefore have been possible. The postcondition and continuation clauses of the subtype function hold immediately on exit and imply the supertype “observations” as expected.

3.5 Histories and Transitions

Object oriented approaches differ from purely functional or logical approaches for at least one important reason: objects persist; they have an identity and a state. Consequently their behaviour is, in general, expected to be different for two subsequent calls of the same function with the same parameters.

This is called *persistent data structures* [6], in which the overall effect of operations matters, functionally and also when reasoning about complexity, rather than that of an individual operation.

Concurrent updates can be captured as sequences of local changes of object spaces. These sequences are sometimes called evolving algebras. For the purposes of this paper it suffices to consider them as graph transformations expressed in terms of a few primitive atomic graph operations. Assuming that dotting $a.b.c$ evaluates from left to right and expressions $m(e_0, \dots, e_{n-1})$ evaluate arguments in arbitrary order, these primitive operations are:

- $v := e$ ($r := x$) assign e (x) to $Self.v$ ($Self.r$);
- $new(v_0 = e_0, \dots, v_{n-1} = e_{n-1})$ creates a new vertex o with $o.type = Self.type$ and $o.v_i = e_i$;
- $del(r)$ deletes the edge $Self \xrightarrow{r}$;
- gc deletes all vertices unreachable from multithread;
- if** C **then** l_1 **else** l_2 changes at to l_1 (or l_2), if condition C hold (does not hold), respectively;
- goto** l changes at to l (unconditional jump);
- call**($a, m, e_0, \dots, e_{n-1}$) creates and runs a new frame object f for the call $a.m(e_0, \dots, e_{n-1})$;
- take**(a, i) tests condition a for value i , blocks on fail, undefines a on success;

make(a, i) sets condition a defined for value i , awakening a blocking thread, if any;

ret returns res and control to caller, possibly spawning off a thread for continuing computing;

output(e) outputs value of expression e ;

stop terminates the current thread and goes into state **finished** if there is no more thread;

error terminates the program with an error.

Above, e is an expression over domain D , $r \in Ref$, $v \in Val$, $a \in A$, $l, l_1, l_2 \in \mathcal{N}$, $x, y \in Ref^*$, and $z \in Ref^* \cup Ref^* Val$, C is a boolean expression.

For simplicity we have only considered a single condition with **take** and **make**. It is straightforward to extend the semantics to include multiple conditions. At any time it is only possible to operate on vertices reachable from the current thread. The gc operation allows us to account for garbage collection of unreachable objects³.

We associate to operations $o \in O$ the mapping $\llbracket o \rrbracket : \mathcal{S}_{D, \Sigma} \rightarrow \mathcal{S}_{D, \Sigma}$, called a *basic state transformation*. In general for each operation o we define $\llbracket o \rrbracket(\mathbf{error}) = \mathbf{error}$.

For the dynamic aspect, we consider *computations* and *histories*. These are finite or infinite sequences of states which can be obtained by the basic operations defined by the program. Transitions in different Kens proceed in parallel. Transitions of different threads in one Ken may commute (subject to delay and mutual exclusion).

Definition 3.5 (Computational Histories)

Let \mathbf{D} be a many sorted algebra; let O be the set of primitive thread operations above; let P be a set of functions; and let $C \in (\mathcal{N} \times \mathcal{O} \times (\mathcal{P} \uplus \{\perp\}))^*$ (called the finite control or program). A computation is a finite or infinite sequence of distributed states $\Gamma = (S_0, S_1, \dots, S_n, \dots)$ satisfying the following properties:

- (i) $S_0 = I$ is the initial state.
- (ii) $S_{i+1} = \llbracket o \rrbracket S_i$ where $S_i = (G_i, f_{ij}.Self, Stack(t_{ij}), PC_i)$, f_{ij} is a thread, and $\langle PC_i, o, x \rangle \in C$.
- (iii) $S_{i+1} = \mathbf{error}$, if $S_i = \mathbf{error}$, if $\langle PC, o, x \rangle \in C$ and o contains an undefined path expression, or if $S_i = (G_i, f_{ij}.Self, Stack(t_{ij}), PC_i)$ and there is no $\langle n, o, x \rangle \in C$ with $n = PC_i$.
- (iv) Γ is finite iff the last state is **finished**.

³Note that intra-Ken garbage can be collected in parallel.

A history H is a computation where all states are different from **error**. The set of all histories of M is denoted by \mathcal{H}_M . A subhistory is a subsequence of H . A subhistory $H' = (S'_0, \dots)$ induced by a function $(m, Par, Loc, mode) \in P$ is a sequence of states different from **error** satisfying property (ii), whose first state is in the set of initial states for m

$$I_M(m) = \{(G, f.Self, Stack(t_f), PC) \in \mathcal{S}_M :$$

$$\exists o \in O : \langle PC, o, (m, Par, Loc, mode) \rangle \in C\},$$

and if the sequence is finite then the last state is a final state for m and initial state $S'_0 = (G'_0, Self'_0, Stack'_0, PC'_0) \in I_M(m)$:

$$F_M(m, S'_0) = \{(G, Self'_0, Stack'_0, PC) :$$

$$\langle PC, \mathbf{return}, x \rangle, x \in \{\perp, (m, Par, Loc, mode)\}\}$$

The set of all subhistories induced by function $(m, Par, Loc, mode) \in P$ is denoted by $\mathcal{H}_M(m)$. The concatenation of two subhistories H_1, H_2 is defined by

$$H_1 \circ H_2 = \begin{cases} H_1 & \text{if } H_1 \text{ is infinite} \\ (S_1, \dots, S_k, S_{k+1}, \dots) & \text{if} \\ & H_1 = (S_1, \dots, S_k) \text{ and} \\ & H_2 = (S_k, S_{k+1}, \dots) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The control associates with a functions in P or the main function \perp an enumeration of primitive operations to execute.

Property (i) defines that a computation has to start with an initial state. Observe that there is no input. In order to model input, we may define *main* with parameters representing the input stream.

Property (ii) states that the next operation whose label is the program counter is executed. If there is no such operation, (iii) defines the computation to be erroneous. (iv) says that a computation is terminating iff **stop** is executed as this is the only operation whose result is state **finished**. Hence **stop** is executed in any computation at most once. Observe that if one of the states is **error** then by (iii) the computation is infinite and all the following states are also **error**. Hence all finite computations are histories. Unless stated differently, we usually assume that a history is a finite computation.

Often, a history (S_1, \dots, S_n) is considered as a graph $S_1 \xrightarrow{o_1} S_2 \rightarrow \dots \rightarrow S_{n-1} \xrightarrow{o_{n-1}} S_n$ where the o_i are the basic state transformations. This justifies the definition of history concatenation above, where a final state S_k of one history is the initial state of another.

4 Conclusion and Future Work

We have described an approach towards a type and reasoning system for concurrent objects. This extends previous work in the Eiffel and Sather context and is likely applicable to other OO languages. We started from Meyers proposal of “separate” computing disentangling its intertwined concepts of distribution, locking and call continuations. In our operational semantics and prototype implementation of cSather we have preserved some fundamental runtime primitives of pSather. However, we have abandoned concurrency control constructs in code altogether and rather promote a strict separation between functional (sequential) code and concurrent interface protocols for Sather.

Our approach aims at provability and efficiency. Achieving such a goal is intrinsically hard in the presence of concurrency due its very high complexity. Besides from the sheer scale of mission-critical systems, this complexity arises from the fact that in concurrent systems many activities interfere, require synchronisation, can independently fail and are subject to hazards such as race conditions. Managing this complexity and meeting increasing quality and safety demands without giving up efficiency has become an important goal in software engineering of distributed objects.

We showed that our semantics preserves substitutability and stability of design-by-contract and thus avoids common forms of the inheritance anomaly. The results are moderate but promising steps towards component oriented methods for concurrent systems construction and analysis. We feel that the results we obtained confirm the need to separate semantic inheritance (specification subtyping) from implementation inheritance (code reuse), especially so in the context of concurrent objects.

Our extensions aim to build on familiar concurrency concepts and reasoning techniques, notably the use of Hoare monitors and Hoare logic to simplify transferring existing skills and technologies into a rapidly growing domain of concurrent and distributed objects. cSather elements have a *deja-vus* look and feel of Hoare monitors and monitor variables, although multiple interfaces and multi-threading are supported.

In an ongoing collaboration we are developing a cSather prototype and tools for presenting and analysing our interface definitions as colored Petri nets. On the theoretical side, we have been working on a formal operational semantics based on the set of atomic transitions. CO protocols for some common concurrency problems are under development together with proofs for their well-behavedness.

Acknowledgements

We gratefully acknowledge support under the ESPRIT grant ECAUS003 and from the Australian Department of Industry Science and Technology. Thanks to A.S.M. Sajeev, Ivan Rayner and Wolf Zimmermann for many stimulating and most fruitful discussions on concurrency, objects and inheritance.

References

- [1] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, Vol. 17 no. 4, pp. 471-522, Dec 1985.
- [2] C. Chambers, D. Ungar and E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Notices*, Vol. 24 no. 10, pp. 49-70, 1989. OOPSLA 1989 Conf. Proc.
- [3] S. Ferenczi. "Guarded Methods vs. Inheritance Anomaly - Inheritance Anomaly Solved by Nested Guarded Method Calls" *ACM SIGPLAN Notices* Vol. 30 no. 2, pp. 49-58, Feb 1995
- [4] J.C. De Kergommeaux and P. Codognot. "Parallel Logic Programming Systems". *ACM Computing Surveys* Vol. 26 no. 3, pp. 295-336, Sep 1994
- [5] S. Matsuoka and A. Yonezawa. "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages" In G. Agha et al. (eds): *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, pp. 107-150, 1993
- [6] K. Mehlhorn and A. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Vol. A*, pp. 301-342. MIT-Press, 1990.
- [7] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
- [8] B. Meyer. *Eiffel - The Language*. Prentice Hall, 1992.
- [9] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, Vol. 36 no. 9, pp. 56-80, Sep 1993.
- [10] S.M. Omohundro. Sather provides nonproprietary access to object-oriented programming. *Computers in Physics*, Vol. 6 no. 5, pp. 444-449, 1992.
- [11] S.M. Omohundro. The Sather 1 Specification. Technical report, International Computer Science Institute, Berkeley, 1993.
- [12] M. Philippsen. Imperative Concurrent Object-Oriented Languages. Technical report, International Computer Science Institute, Berkeley, 1995.
- [13] A.S.M. Sajeev and H.W. Schmidt. *TOOLS'94 Tutorial: Object-oriented Concurrent Programming, Technology of Object-Oriented Languages and Systems (TOOLS) Pacific '94 Conference*, Melbourne. Monash University, 1994.
- [14] H.W. Schmidt and S. Omohundro. *Object-Oriented Programming: The CLOS Perspective*, chapter 'CLOS, Eiffel and Sather: A comparison', pp. 181-213. MIT Press, 1993.
- [15] H.W. Schmidt and R. Walker. TOF - an efficient type system for objects and functions. Technical Report TR-CS-92-17, Department of Computer Science, The Australian National University, Nov 1992.
- [16] H.W. Schmidt and W. Zimmermann. A complexity calculus for object-oriented programs. *Object Oriented Systems*, Vol. 1 no. 2, pp. 117-147, 1994.
- [17] H.W. Schmidt and W. Zimmermann. Reasoning about complexity in object-oriented programs. In *Proceedings of IFIP Working Conference on Programming Concepts, Methods and Calculi (PRO-COMET 94)*, San Miniato, Italy, pp. 541-560, 1994.
- [18] A.U. Shakar. An introduction to assertional reasoning for concurrent systems. *Computing Surveys*, Vol. 25 no. 3, pp. 225-302, 1993.
- [19] C. Szyperski, S. Omohundro, and S. Murer. Engineering a Programming Language: The Type and Class System of Sather. Technical report, International Computer Science Institute, Berkeley, 1993.
- [20] P. Wegner. "Tradeoffs between Reasoning and Modelling". In G. Agha et al. (eds): *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, pp. 22-41, 1993