2008

Practical Strengthening of Preconditions

Ashish Kundu

Patrick Eugster
Purdue University, p@cs.purdue.edu

Report Number:
08-029

Practical Strengthening of Preconditions

Ashish Kundu
Patrick Eugster

CSD TR #08-029
November 2008

Practical Strengthening of Preconditions

Ashish Kundu[1] and Patrick Eugster[1]

Department of Computer Science, Purdue University
[ashishk, peugster]@cs.purdue.edu

Abstract. This paper takes a closer look at behavioral subtyping in the context of concurrency, by considering an example of subtyping of a concurrent data-structure taken from sensor networks. Akin to the extension of a state-machine, this example illustrates conflicts caused by the interference of concurrency and inheritance. In short, the extension consists in declaring additional fields in a subclass, thus extending the state space of the super-class. Reminiscent of a famous illustration case for inheritance anomalies, the extension leads to violating behavioral subtyping by requiring the strengthening of preconditions in the subclass to account for the extended state space. We provide a solution to this bottleneck by characterizing cases in which such a strengthening can safely occur. To that end, we dissect preconditions according to various criteria, such as their semantics (wait or "traditional" correctness), or the objects they involve (method arguments or fields of the receiving object). Using denotational semantics, we formally prove that such strengthening of preconditions through wait conditions is safe nd does not break the contract. We implement our relaxation in Eiffel, and illustrate how it preserves modular reasoning – in particular about deadlocks.

1 Introduction

Application of axiomatic semantics [13] in the context of object-oriented programming languages with subtype polymorphism has been triggered through the seminal work on *behavioral subtyping* by Liskov and Wing [20]. Significant advances for the modular reasoning about object-oriented programs have been achieved in combination with explicit language syntax in the context of programming languages such as Spec# [21] and Eiffel [24], or with annotations in the Java Modeling Language (JML) [4]. Recent refinements to behavioral subtyping including for example investigations around consistency and ownership [18], or the tracking of null references [10]. With the object-oriented approach being widely used for the development of industrial-scale software, it is important to develop reliable trusted components. First steps in direction have been taken to benefit from the behavioral reasoning beyond the scale of single components [1].

However, most industrial-scale software currently being developed is not only considerably large — thus benefiting from modular development, reasoning, and proving methodologies — but also strongly *concurrent*. This trend can be expected to even increase given the current proliferation of multi-core architectures. Yet, concurrency still bears many challenges for modular and behavioral approaches, and has been only considered late, while quite ironically, a classical example for behavioral subtyping is the

bag [20], which by definition expresses the presence of multithreading. In this paper, we use *buffers* - a slight variation of *bags* for illustration purposes.

Initially, concurrency was dealt with by (1) making strong *assumptions*, such as the availability of transactional support to execute methods in a (seemingly) atomic manner, i.e., without interference with any other methods, and by (2) introducing strong *constraints*, for instance allowing the subtype T_2 of a type T_1 to introduce a new method M only if that method is expressible with existing methods of T_1 [20]. Several methodologies have appeared to relax constraints of type (1) in the context of axiomatic semantics (e.g., [29, 16]), i.e., to allow for more parallelism. Similar advances are being made in object-oriented settings, as illustrated by [15, 32]. Restriction (2) above is still very compelling, and has been motivated by modular/incremental proving. Together with the widely accepted constraint that a subtype T_2 of a type T_1 M of type T_1 can neither strengthen the precondition nor weaken the postcondition of any method$M$ defined in T_1, one can straightforwardly reason modularly about correctness. That is, no client — or more generally no program — making use of type T_1, when an instance of T_1 is substituted by one of T_2 can "behave unexpectedly" by as by blocking or yielding an un-anticipatable exception. Behavioral subtyping further reduces opportunities for code reuse.

Given a correct deadlock-free program with clients accessing a shared object of type T_2 (or T_1) only through variables of type T_1, one can now moreover extend the program by adding clients accessing the object through variables of type T_2 – with only additional reasoning required for these new clients to prove (further) absence of deadlocks. In short, if the new method M is correct and is only composed of previously available methods of T_1, which are used in a sequence which describes a possible schedule for T_1, then their respective preconditions, under consideration of the sequence, will yield a precondition for M which only is true if the sequence can be executed, avoiding the introduction of a deadlock after partial or complete execution of M. This constraint provides very strong guarantees, but is also quite restrictive, and leaves the problem of bootstrapping open: no type with a single method can be used to start incrementally proving deadlock-freedom. More complex types/programs must be manually proven to be deadlock-free for bootstrapping.

This paper revisits behavioral subtyping in the context of concurrency, and proposes a relaxed model. constraints such as the above. We first start from the buffer example of [20], modeled in Eiffel, and extend it in a way currently impossible according to the traditional confines of behavioral subtyping, yet relevant in many practical scenarios. More precisely, we take a look at scenarios in which shared data-structures are extended by augmenting their state space, leading to *strengthening* preconditions of ancestor classes and thus violating behavioral subtyping. These scenarios are typical of state machine extensions, and similar to one of the scenarios used in [22] to illustrate *inheritance anomalies*; the bottleneck observed confirms the claim of [6] that indeed useful and sensible scenarios are ruled out by the traditional notion of behavioral subtyping. More precisely, if we go ahead and try to implement a given, indeed sensible, buffer subtype through a subclass, we end up violating behavioral subtyping.

To convey our claims, we take a closer look at the notion of *precondition* under the perspective of a "contract between client and supplier objects" [23], as well as by con-

sidering its similarity with *guarded commands* in the context of multithreading, arguing that in certain cases, it might be useful and safe to *strengthen* preconditions, which is an intriguing thought as it goes against any common belief: "*... Strengthening the precondition, or weakening the postcondition, would be a case of "dishonest subcontracting" and could lead to disaster...*" [8]. To that end, we dissect preconditions according to different dimensions, such as the *origins* of the objects involved (method arguments versus fields of the receiver object), their *mutability* (immutable versus mutable objects), and their *evaluation semantics* (eventual versus immediate satisfaction), pointing out that an eventually satisfied condition (a "wait condition") expressed solely on mutable fields of the recipient added by the subclass, along with some additional constraints, can be *conjoined* with a precondition of the super-type for the same method without yielding unexpected exceptions and without leading to incorrect behavior. The intuition behind this is that preconditions have a twofold purpose: (1) they express requirements for clients with respect to the arguments passed upon calls to respective methods, and (2) they represent an implicit (potential) synchronization barrier, which may rely on internal synchronization variables of the receiver. We then demonstrate that, when abiding to the set of conditions we state, and assuming some minimal runtime support which is more moderate than the assumptions used by other relaxations of behavioral subtyping (e.g. [7]), it is still possible to prove deadlock-freedom for programs in the modular way outlined above, i.e., programs initially making use of an object of type T_1 can be safely changed to make use of an instance of a subtype T_2 of T_1, and the addition of clients accessing the object through variables of T_2 only requires additional proofs of deadlock-freedom involving those clients. We prove such safety property through denotational semantics. Our extensions make the explicit encoding of sequences in a way similar to state machines easier for concurrently accessed objects in behavioral subtyping scenarios.

Roadmap. Section 2 presents the problem scenario outlined above. Section 3 dissects preconditions according to different criteria. Section 4 outlines rules for subtyping building on our model of preconditions. Section 5 describes the denotational semantics and proof for safety for strengthening of preconditions using *upon* preconditions. Section 6 presents how to retain modular reasoning about deadlocks. Section 7 presents the implementation in Eiffel. Section 8 discusses the problem and the proposed solution of strengthening preconditions with *wait* semantics. Section 9 summarizes related work. Section 10 draws final conclusions.

2 Problem and Solution Outline

In this section we illustrate a considerable restriction of behavioral subtyping in the context of multithreading through a simple buffer implemented in the Eiffel language. The problem is however of a general nature, and applies for instance also to the Java Modeling Language (JML) [4, 17].

2.1 A Buffer

Consider the buffer outlined in Listing 1 implemented in the Eiffel language [24] (syntax simplified in the following). In a nutshell, Eiffel is a pure object-oriented programming language with multiple inheritance. Types are implicitly defined by classes. Eiffel furthermore provides genericity, as illustrated by the `Buffer` class, which introduces a type parameter `G` for the type of its elements. The *feature* (attributes or methods in Eiffel parlance) `count` represents the number of elements currently in the buffer, and `max` is a constant which acts as upper bound for that number.

The meaning of `separate` will be detailed later-on. At this point it is sufficient to know that it is used to denote entities whose use might require synchronization of concurrently executing threads. Liskov and Wing's example in [20] assumes `get()` and `put()` to be executed atomically, by means of a transactional mechanism.

Figure 3 outlines the use of two such buffers in the implementation of a sensor device, inspired by [14]. The medium access control (MAC) layer provides functionalities to send and receive messages; the sending component simply attempts to extract messages from the output buffer, and the receiver component adds any messages to the input buffer. The application interacts with the network through these buffers. Both the application as well as the MAC layer are programmed against the `BUFFER` interface. For obvious reasons they can not be easily modified.

2.2 Adding Explicit Temporal Constraints

Now consider the case of a *global* congestion control mechanism being added [14, 30] in which, when a given nodes' output buffer fills up, the next message drawn from it will be appended a corresponding notification. Upon reception of such a notification from a node previously not known to be congestioned, any node's input buffer increases a counter, which, once exceeding a given threshold, to inhibit both the MAC layer as well as the application in order to avoid further sends by restricting the addition or removal of elements from the output buffer. Listing 2 represents the extended output buffer.

The trained eye will immediately notice that this example violates behavioral subtyping, as the preconditions of both `put()` and `get()` are strengthened through the *not congestioned* predicate.[1]

This example is very similar to the *lockable buffer* used to illustrate *inheritance anomalies* [22]. Though these have been shown to be more due to the interplay of subtyping and inheritance mechanisms rather than concurrency and inheritance mechanisms, the used examples represent valid specialization scenarios.

The lockable buffer is described as a subtype of both `BUFFER` and `LOCKABLE_BUFFER` (see Listing 4). Inheriting from the latter type adds the restriction that any methods become inaccessible if `locked` is false. This leads to extending preconditions of all methods with a corresponding check.

The examples provided here make use of "internal explicit state tracking" within a buffer subtype. We refrain at this point from using any extraneous "state machine" as

[1] To avoid this, Eiffel by default disjoins the precondition of a given method redefined in a subclass with the precondition in the super-class.

```
class BUFFER[G]
feature [...]
  count: INTEGER -- total number
        of elements
  max: INTEGER -- maximum size
       of buffer

  get: separate G is
  require
    count > 0
  do
    [...]
  ensure
    count = old count - 1
  end -- get

  put(element: separate G) is
  require
    count < max
  do
    [...]
  ensure
    count = old count + 1
  end -- put
end -- BUFFER
```

Fig. 1. Bounded buffer in Eiffel

```
CONG_CTRL_BUFFER[G] inherit BUFFER[G]
redefine put, get
feature [...]
  congestioned: BOOLEAN := False

  congestion(new_state: BOOLEAN) is
  require
    congestioned = not new_state
  do
    congestioned := new_state
  ensure
    congestioned = new_state
  end -- congestion

  redefine get: separate G is
  require
    count > 0 and not congestioned
  do
    [...]
  ensure
    count = old count - 1
  end -- get

  redefine put(element: separate G) is
  require
    count < max and not congestioned
  do
    [...]
  ensure
    count = old count + 1
  end -- put
end -- CONG_CTRL_BUFFER
```

Fig. 2. Buffer for global congestion control in Eiffel

found for instance in *active objects* (e.g., [3]) to model temporal ordering of methods, or from temporal logic-like specifications, as the latter ones can easily lead to prohibitive complexity (see Section 9.1). The reason is that the basic buffer itself does not use such mechanisms. Explicit mutual exclusion locks, though possible in the portrayed cases, make the modeling of more advanced constraints on temporal ordering of methods very tedious (see Section 8.2).

2.3 Solution Outline

In the following we describe a solution implemented in the context of Eiffel which allows for such extensions, and in fact much more powerful ones. In short, it relies on (1) a precise characterization of possible extensions and a corresponding set of restrictions which are underpinned by a new keyword used in preconditions, and on (2) runtime support.

Listing 4 presents the solution for the lockable buffer based on this syntax: our new `upon` keyword announces a precondition that is conjoined with that of the super-class for the same method.

Our relaxation of behavioral subtyping roughly consists in allowing a class to extend the *state space* of an ancestor class by adding fields, and strengthening preconditions

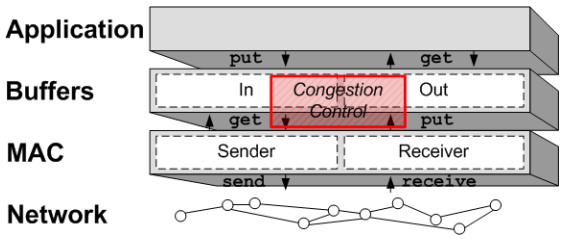


Fig. 3. Adding global congestion control

```
LOCKABLE_BUFFER[G]
inherit BUFFER[G], LOCKABLE
redefine put, get

feature redefine get: G is
require
  upon not locked
do
  Result:= Precursor
end -- get

redefine put(element: separate G) is
require
  upon not locked
do
Precursor(element)
end -- put
end
```

Fig. 4. Lockable buffer in Eiffel

```
deferred class LOCKABLE
feature locked: BOOLEAN := False
lock is
require
  not locked
do
locked:= True
ensure
  locked
end -- lock

unlock is
require
  locked
do
  locked:= False
ensure
  not locked
end -- unlock
end -- LOCKABLE
```

Fig. 5. Super-type for lockable data structures

by adding conditions with *wait* semantics expressed on these additional fields. To respect behavioral subtyping, it has to be ensured that these additional wait conditions do hold *at least* in the *safe state* space of such a subtype; this particular state space represents the state subset of the extended state space which coincides with the original state space of the super-class. Though a state *space*, we refer to it in the following simply as the safe state. In the `CONG_CTRL_BUFFER` and `LOCKABLE_BUFFER` examples the original state spaces are extended by `congestioned` and `locked` respectively, and the corresponding safe states are characterized by a value of `False` for the respective fields.

In short, conditions with *wait* semantics simply *delay* clients; extending preconditions by *correctness* conditions could lead to unexpected exceptions being raised in case those conditions are violated and thus to broken semantics. Reasoning can be decomposed into subsets of clients accessing such a shared data-structure through variables of the same type respectively. By ensuring that the safe state is eventually reached, the conditions following the `upon` clauses are trivially satisfied, boiling down to a disjunc-

tion of the original precondition and a wait condition involving all states in the extended space allowed by the `upon` clause modulo the safe state (see Section 8.1).

Similar state machine-like extensions have been described in non-object settings [33], under simpler assumptions (see Section 9).

3 Basic Notations and Model

3.1 Model

We focus on types, which contain both method and field declarations. Methods involve pre- and postconditions as part of their specification, but their bodies are not relevant at this point.

Declarations most commonly consist in tuples, which are denoted by capital letters, e.g., X. Names of such instances are denoted by corresponding lowercase letters e.g. x, and sets of such declarations are denoted for example as $\mathcal{X}$.

$P \in Programs$
$T \in \Gamma : Types$
$I \in \mathcal{I} : Invariants$
$M \in \mathcal{M} : Method\ names$
$F \in \mathcal{F} : Fields$
$X \in Expr : Expressions$
$o, o_i, o_j \in Objects$
$n \in N : Numerals$
$t \in TNames : Type\ names$
$\mu, \mu_i, \mu_j, \nu \in MNames : Mutable\ field\ names$
$f \in INames : Immutable\ field\ names$
$g \in Arguments : Arguments\ to\ a\ method$
$E \in \mathcal{E}$: Boolean eventual conditions in *upon* context
$C \in \mathcal{C}$: Boolean conditions not used in an *upon* context
$B, A \in \mathbb{B}$: Boolean conditions

Fig. 6. Domains

3.2 Dissecting Preconditions

Immediate conditions: An *immediate* condition, denoted $\downarrow C$, occurring in the precondition of a method M in type T is a predicate C that must evaluate to TRUE *immediately* upon an invocation of M on an instance of T for successful execution of the method M. Such a condition corresponds to the "traditional" view of preconditions as correctness conditions; when used solely for program verification it must be proven to hold at a given invocation, or, when combined with runtime support, can yield an exception (which the caller must be prepared for) if not satisfied.

$P ::= \{T\}\ [\{X\}]$
$T ::= (t, \{M\}, \{F\}, \{\mathrm{I}\}, [\{t\}])$
$M ::= (B, S, A)$
$S ::= (t, GL)$
$GL ::= [\{(g, t\}]$
$F ::= (f, t) \mid (\mu, t)$
$I ::= C$
$A ::= C$
$B ::= C \mid U$
$U ::= \textbf{upon}\ E$
$Z ::= f \mid g \mid n$
$C ::= E \mid f\ R\ Z \mid f\ R\ \nu \mid C\ D\ C \mid \textbf{not}\ C$
$D ::= \textbf{and} \mid \textbf{or}$
$R ::= < \mid > \mid \textbf{==} \mid <= \mid >=$
$E ::= \mu\ R\ Z \mid \mu_i\ R\ \mu_j \mid \textbf{not}\ E$
$X ::= t\ o \mid o := \textbf{new}\ t \mid o_i := o_j \mid o.M(OL) \mid o.M() \mid o.\mathit{setsafe}()$
$(* \mathit{omitted} : if - then - else, while *)$
$OL ::= \{o\}$

Fig. 7. Abstract Syntax

Eventual conditions: An *eventual* condition, denoted $\diamond C$, appearing in the precondition of a method M in type T is a predicate C that must evaluate to TRUE *eventually* for successful execution of the method M. Such a condition is similar to a guard, and reflects the `when` clause of JML, or the `separate` keyword in Eiffel (see Section 6.1).

The distinction between eventual and immediate conditions in the context of preconditions allows for increased flexibility, in the sense that eventual conditions can be strengthened in some contexts that we will characterize shortly. Immediate conditions can not strengthened, because their runtime monitoring can lead to exceptions being thrown to callers, which might be catered for these exceptions if they represent additional constraints, which they were not aware of. These constraints thus correspond to the original interpretation of preconditions as contracts between consumers and suppliers [23].

3.3 Mutability

Key to the distinction of eventual and correctness conditions is the notion of object mutability. In short, a predicate expressed solely on immutable objects is naturally interpreted as an immediate condition: if it does not hold immediately upon invocation of a method it will never hold. Anything else is an eventual precondition. We hence distinguish between *mutable* and *immutable* objects — more precisely actual arguments — *with respect to a given condition*. By safe we assume mutability of objects. The distinction then depends on several factors, as outlined below and summarized by Figure 8:

Type mutability: An argument of a condition can be immutable by its very *type*. Examples, where the type declaration has an effect are for instance primitive types or

their corresponding object types as `boolean` or `Boolean` in Java respectively, whose instances are all invariably immutable. Eiffel proposes similarly *value types*, which can also be custom-defined. A type-immutable argument is thus a *formal argument* of an immutable type. We assume that a corresponding *actual argument* can never become mutable: according to the original definition of behavioral subtyping [20], a mutable subtype of an immutable type would violate *history conditions* which can only be strengthened by subtypes and thus further narrow down the possible states for objects.

Context mutability: An argument of an a priori mutable type can still become immutable in a given *context*. By context we mean the way it is declared. This can occur for instance through additional keywords prefixing a formal argument in a given language, e.g., `const` in C++, `final` in Java. Further, `pure` in JML can be used to specify if the associated method does not modify any of the arguments, which means that such arguments are effectively not mutable in such a context. We do however not consider the case where one can define an immutable type as subtype of a mutable type (see Section 9.2). precise). Hence, the dynamic type of an actual argument can not render a type-mutable argument context-immutable. This is both a reasonable and useful assumption, which results in that mutability/immutability of a given argument within a predicate occurring within a precondition is entirely statically determinable.

In the following, a variable $\star V$ denotes a mutable variable if $\star{=}\oslash$ (e.g. $\oslash V$), or an immutable one if $\star{=}\infty$ (e.g. ∞V). If $\star$ is unknown or irrelevant it will be omitted.

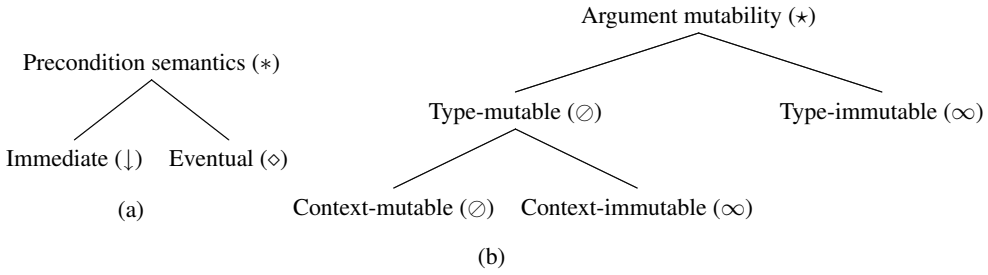


Fig. 8. Classification of (a) preconditions and (b) precondition arguments.

4 A Relaxed Notion of Behavioral Subtyping

Based on the above definition of conditions, we are able to define a relaxed notion of behavioral subtyping, in which preconditions can be strengthened. Figure 10 outlines to that end two notions of subtyping, the first one of which corresponds to the "traditional" behavioral subtyping rule ($<:_T$), and the one introduced here ($<:_S$).

In our definition, the precondition of the method of a supertype T_1 can be strengthened in the subtype T_2 through use of a new eventual condition expressed on additional

mutable fields of the subtype. If, modulo the parts retained from the rule of traditional $<:_T$, all conditions by which preconditions of a supertype are extended hold for some well-defined safe state, then T_2 is a valid subtype of T_1 according to $<:_S$. For specification purposes, the new eventual condition is annotated with a keyword `upon` preceding it and is referred to as *upon precondition*.

```
class CONG_CTRL_BUFFER[G] inherit BUFFER[G] redefine put, get

feature
  [...]
  congestioned: BOOLEAN := False

congestion(new_state: BOOLEAN) is
  [...]

redefine get: separate G is
require
  upon not congestioned
do
  [...]
ensure
  [...]
end -- get

redefine put(element: separate G) is
require
  upon not congestioned
do
  [...]
ensure
  [...]
end -- put
end -- CONG_CTRL_BUFFER
```

Fig. 9. Buffer for global congestion control with upon conditions

4.1 Upon Conditions

An *upon* condition is an eventual condition that involves one or more *elementary* conditions and is preceded by a keyword `upon` in a precondition of a method (Listing 4). Each such *elementary* condition involves one or more single mutable variables introduced in the type. It is sufficient to note that besides variables, these can involve literals and binary operators on these, accesses to fields of variables (which are accessible at that point), or method invocations on variables, with further variables or expressions used as parameters. The upon conditions of a method M can only refer to the internal state of the object. For instance, it must not refer to parameters of M. An upon precondition locally introduced in T for method M is visible to all the subtypes of T, to which M is also visible.

upon(M, T): Upon precondition of a method M in type T denoted by *upon*(M, T) is defined recursively as a conjunction of the upon condition for M introduced locally

in T and the upon condition introduced for the method in each supertype of T that M redefines. Let *local-upon*(M, T) refer to the upon precondition of M locally introduced in T. Let *uponset*(M, T) be the set of *upon* conditions introduced in each of the supertypes of T(inclusive of T).

$$\textit{uponset}(\tilde{M}, \tilde{T}) = \textit{local-upon}(\tilde{M}, \tilde{T}) \bigcup_{T=supertype(\tilde{T})} \textit{uponset}(M, T). \quad (1)$$

where $\tilde{M} \in \mathcal{M}(\tilde{T})$ redefines $M \in \mathcal{M}(T)$.

$$\textit{upon}(\tilde{M}, \tilde{T}) = \bigwedge_{u \in \textit{uponset}(\tilde{M}, \tilde{T})} u \quad (2)$$

If $\tilde{T}$ does not introduce any upon precondition for $\tilde{M}$ locally,

$$\textit{upon}(\tilde{M}, \tilde{T}) = \textit{upon}(M, T) \quad (3)$$

For expository purposes, we assume the absence of superfluous variables and ill-typed conditions in an upon condition and that an upon condition is represented in CNF (Conjunctive Normal Form). If a (mutable) variable is passed to a method within a precondition, it is actually used, and no bogus conditions (e.g., v == v) appear. Methods involved in preconditions are considered to be devoid of side-effects (a common assumption not stated explicitly). In CNF representation, the elementary condition in an upon condition is a clause and thus involves one or more mutable variables.

Upon conditions are closely associated with a notion of safe-state of objects. In the example related to sensor networks (Figure 3), the sender in the MAC layer is aware of only the interfaces - `get()` and `put()` and the contracts for these methods as defined in `BUFFER`. However the implementation of these interfaces are provided by the congestion control buffer (`CONG_CTRL_BUFFER` in Listing 9). Any strengthening of preconditions of the `get()` and `put()` methods should be safe with respect to the MAC layer client. In general, such strengthening in a type T should be safe for a client of any of the supertype of T. In a concurrent setting, an object of T maybe accessed by multiple clients, each independently assuming it as an object of T or some supertype of T. In our example, the application layer and the MAC layer are the two clients assuming the buffer to be an object of `CONG_CTRL_BUFFER`(T) and `BUFFER` (supertype of T), respectively. In order to ensure that "nothing goes bad", we introduce the notion of *safe state* of an object of a type with respect to a given supertype.

4.2 Safe State and Mutable Fields

In order to honor the behavioral subtyping and the associated subtype requirement [20] the upon conditions satisfy the following requirement: an added eventual condition is satisfied by some special values of the additional fields. We will henceforth refer to these values as *safe* values. For a given field F, it is denoted ∇F. These additional fields are initialized to these safe values during the object construction. In short, such a value enables a behavioral subtype to function as an instance of the supertype in a "safe state" (or safe mode), which is the absence of any (non-terminating) actions defined through

additional methods of the subtype on the additional fields. In the example (Listing 4), the safe value of the added field `locked` is FALSE. If this value is adopted, the state space of a `LOCKABLE_BUFFER` is reduced to that of a `BUFFER`, making it behave exactly as an instance of that latter type.

The additional fields that are of importance to upon conditions are mutable fields, since evaluation of an eventual condition (upon condition) waits on its mutable fields to hold values that collectively make the condition TRUE. A collection of such values of the mutable fields are statically defined and it refers to the safe state. In what follows, we define the safe state (safe mode) in terms of the mutable fields and their safe values.

Mutable fields: *mutable*(M, T) denotes the set of the mutable fields used in the *upon* precondition of M - *upon*(M, T).

$$\textit{mutable}(M,T) \triangleq \{\oslash F \mid F \in \mathcal{F}(\textit{upon}(M,T))\}. \tag{4}$$

For each field $F \in$ *mutable*(M, T), *safe* value ∇F makes each clause in *upon*(M, T) involving F, TRUE. Let *mutable*(T) be defined as the set of all mutable arguments in a type T used in all its upon preconditions:

$$\textit{mutable}(T) \triangleq \bigcup_{M \in \mathcal{M}(T)} \textit{mutable}(M,T). \tag{5}$$

In order to ensure that o is in safe state, all the upon conditions (for all methods) in T_2 excluding those defined in T_1 and its supertypes must hold. Thus a safe state for objects of T_2 against T_1 is defined over the *safe* values of all the mutable fields used in all such upon preconditions - all fields in (*mutable*(T_2) $\setminus$ *mutable*(T_1)).

Safe state: A safe state of an object o of type T_2 with respect to its supertype T_1 is a state that makes all the strengthened conditions hold that are introduced by each and every supertype that is either T_1 or a subtype of T_1. *safestate*(T_2,T_1) denotes such a state. Let $[\![C]\!]s$ denote the evaluation of C on an object which is in state s.

$$\textit{safestate}(T_2) \triangleq \{s \mid [\![\bigcup_{M_2 \in \mathcal{M}(T_2)} \textit{upon}(M_2,T_2)]\!]s = \text{TRUE}\} \tag{6}$$

$$\textit{safestate}(T_2,T_1) \triangleq \{s \mid [\![\bigcup_{M_2 \in \mathcal{M}(T_2)} (\textit{local-upon}(M_2,T_2) \wedge \textit{upon}(\tilde{M},\tilde{T}))]\!]s = \text{TRUE}\} \tag{7}$$

where T_2 is a subtype of $\tilde{T}$, which in turn is a strict subtype of T_1 ($\tilde{T} \neq T_1$), and $\tilde{M} \in \mathcal{M}(\tilde{T})$ redefines M_2.

$$\forall s \in \textit{safestate}(T_2,T_1) : s \triangleq \bigcup_{F \in (\textit{mutable}(T_2) \setminus \textit{mutable}(T_1))} \{(F, \nabla F)\} \tag{8}$$

We will see in the following section in more detail how and where the safe value comes to play.

[INVARIANTS]
$\sigma, \omega \in \Gamma;\ \Gamma \vdash \mathcal{I}(\omega) \Rightarrow \mathcal{I}(\sigma)$

[FIELDS]
$\Gamma \vdash \forall F_i = \langle f_i, T_i \rangle \in \mathcal{F}(\sigma)\ \exists\ F_j = \langle f_j, T_j \rangle \in \mathcal{F}(\omega)\ \ni\ (f_i = f_j) \wedge (T_i = T_j)$

[CONTRAVARIANCE OF METHOD ARGUMENTS]
$\Gamma \vdash \forall M_i = \langle m_i, B_i, S_i, A_i \rangle \in \mathcal{M}(\sigma)\ \exists M_j = \langle m_j, B_j, S_j, A_j \rangle \in \mathcal{M}(\omega)\ \ni$
$\forall G_{i.k} \in \mathcal{G}(S_i), G_{j.l} \in \mathcal{G}(S_j)\ (l = k) \Rightarrow T(G_{i.k}) <:_S T(G_{j.l})$

[COVARIANCE OF METHOD RETURN]
$\Gamma \vdash \forall M_i = \langle m_i, B_i, S_i, A_i \rangle \in \mathcal{M}(\sigma)\ \exists M_j = \langle m_j, B_j, S_j, A_j \rangle \in \mathcal{M}(\omega)\ \ni$
$T(S_j) <:_S T(S_i)$

[POSTCONDITIONS]
$\Gamma \vdash \forall M_i = \langle m_i, B_i, S_i, A_i \rangle \in \mathcal{M}(\sigma)\ \exists M_j = \langle m_j, B_j, S_j, A_j \rangle \in \mathcal{M}(\omega)\ \ni$
$A_i \Rightarrow A_j$

[PRECONDITIONS]
$\Gamma \vdash B_j = B_i \wedge \diamond C_j(\mathcal{F}_{\sigma \setminus \omega} \subseteq \mathcal{F}(\sigma) \setminus \mathcal{F}(\omega))\ \ni$
$\forall \oslash F = \langle \mu, t \rangle \in (\textit{mutable}(\sigma) \setminus \textit{mutable}(\omega)),\ (\mu, \nabla\mu) \in s \in \textit{safestate}(\sigma, \omega)$

[RELAXED BEHAVIORAL SUBTYPE]
$$\frac{\Gamma}{\omega <:_S \sigma}$$

Fig. 10. Subtyping rule

4.3 Subyping Rule

The typing rule for the relaxed notion of behavioral subtyping is different than that of the general notion in the definition of the precondition (Figure 10). Typing rules for invariants, fields, method arguments and return, and postconditions are same as that of the general behavioral subtype. For mutable fields, the same rule ([Fields]) applies (with f_j being replaced by μ_j).

From the structure of the precondition rule, it is not hard to infer if a subtype is a relaxed behavioral subtype or not.

4.4 Semantic Rules

We distinguish between the clients based on the type through which they are accessing the object. Let γ is a relaxed behavioral subtype of σ and γ uses a new *upon* condition in the precondition of a method M, which is already defined in σ. Let ω be a relaxed behavioral subtype of γ such that it defines another new *upon* condition for M. Let an object o be instantiated from the type ω. A *subtype client* is one that invokes a method M on an object through a type such that the method M has one or more *upon* defined in that type or its supertype (such as γ or ω). All other clients are *supertype clients*: one

that invokes a method M on o through a type that does not have any upon condition defined (locally or in a supertype).

Intuitively, the requirements outlined previously support the safe use of an object o — despite certain strengthened preconditions — as an instance of any supertypes. The added precondition being a pure wait condition makes it impossible that a client accessing such an object o through a variable of a supertype receives an unanticipated exception. The presence of a safe state with respect to a given supertype in which o behaves just like an instance of that type provides the possibility for such an object to be used exactly like an instance of the given supertype. The semantic rules are as follows:

Construction: A newly constructed object is initialized to a safe state, if supertype clients may access this object.
Evaluation: The *upon* condition in a precondition is evaluated first and upon its evaluation to TRUE, the immediate condition is evaluated, both these evaluations being carried out in an atomic manner.
History Constraint: A subtype client always returns the object to a safe state before relinquishing its mutually exclusive access to the object.
Mutual Exclusion: Supertype clients are never granted mutually exclusive access to an object, while it is not in a safe state in terms of the supertype.

If the object is (going to be) accessed only by subtype clients, a subtype client need not return the object to a safe state before relinquishing its mutually exclusive access to the object. Ensuring that o eventually reaches that state (and/or possibly is initialized in that state) relies on the application scenario. In principle nothing prevents such objects, once exiting the original state space, to never return. This leads to defining additional *history invariants* in the sense of [20]. In most cases however, it is reasonable to assume that a relaxed subtype provides additional methods to transit back and forth between the original (safe) and extended space. In the case of the congestion control buffer, this possibility is provided by switching from contended to not contended state.

Avoiding deadly blocking in combination with mutual exclusion on such objects becomes however a concern. In short, runtime support is required to ensure that a supertype client accessing o, does not obtain mutual exclusion on that object while it is *not* in the safe state with respect to that type. This is discussed in the following section.

4.5 Informal Reasoning

In this section, we reason about the correctness properties of the relaxed notion of the behavioral subtype - in the context of the typing and semantic rules.

The concern is: If a supertype client (type: σ) attempts to access an object o of a subtype (type: ω), will it ever get blocked by invoking a method M, even if it satisfies the precondition $B(M,\sigma)$? The following lemma proves that it will never block, provided the semantic rules are followed in conjunction with the typing rule. A subtype client would obviously encounter the *upon* condition because it is aware of the contracts.

Lemma 1. *A supertype client will never block on an object of a relaxed behavioral subtype (of the supertype).*

Proof. Let the supertype and the subtype be σ and ω, respectively. Let o be an object constructed from ω. Let M be a method defined in σ, which is redefined as $\tilde{M}$ in ω. If $\tilde{M}$ does not involve any local *upon* condition in its precondition, the supertype client will never block. If it involves a local *upon* condition, then we have the following cases.

Case-I: After its construction, o has not been accessed by any client before the supertype client attempts to access it; thus it is in a safe state with respect to all the supertypes of ω (Construction rule). Thus the supertype client would not block on this access.

Case-II: Only supertype clients are accessing the object. Since only subtype clients are aware of the *upon* conditions and only such clients would be able to change the state of the object from a safe state to an unsafe one with respect to this local *upon* condition, the object would never enter any unsafe state. By the Construction rule and the Mutual Exclusion rule, the supertype clients would never block on this *upon* condition.

Case-III: A mix of supertype and subtype clients are accessing the object. The supertype client would not be allowed to access the object if it is in an unsafe state (Mutual Exclusion rule). A subtype client that has access to the object at a given point of time, would always return the object to a safe state (History Constraint rule). A supertype client only waits for a finite amount of time depending on the application scenario before being able to access the object. Therefore, if a supertype client tries to access o after a subtype client has already accessed o by invoking M on o, then the supertype client would not be given mutual exclusion if another client is holding exclusive access to the object o. As per the history constraint rule, a subtype client would release such mutual exclusive access in a finite number of execution steps (finite time). Thus the supertype client would eventually acquire mutual exclusion on the object in a safe state.

Further, a supertype client does not have to carry out any operations to change the state of the object to a safe state. Thus the lemma is proved. QED

5 Denotational Semantics

In order to simplify the exposition (and given the space limits), we propose the denotational semantics [34] of the language relevant to the relaxed behavioral subtyping based on the abstract syntax in Figures 6 and 7. Figure 11 specifies the semantic algebra for the denotational semantics of the strengthening of preconditions through *upon* conditions. Figure 12 specifies valuation functions for a selected set of domains.

The operation ($\_ \rightarrow \_ \,\square\, \_$) behaves as an if-then-else statement [34]. It takes a boolean value as a first argument and if it is TRUE then the second argument is evaluated, else the third one is evaluated. We introduce a notion of a unit of atomicity, which is a sequence of execution steps that is never interrupted. A unit of atomicity is denoted by a bar over the operation under evaluation with a unique identifier (such as α). For example, $E[\![\overline{M}^{\alpha}]\!](o)$ means M is invoked (evaluated) on o in a mutual exclusive access guaranteed by the atomic unit αİn Our specific interest lies in analyzing the semantics of method evaluation in the context of *upon* preconditions. A set of safe objects (*SafeObjects*) is defined to highlight that objects from from safe to unsafe and unsafe

to safe sets. *setsafe* is an operation implemented by types in order to facilitate changing the state of an object to a safe state with respect to each and every strengthened precondition introduced locally and in its supertypes. A newly constructed object is a safe object; the valuation function for $E[\![o := \mathbf{new}\ t]\!]$ is meant to simply highlight such a semantics. (We have precluded use of generators and detailed formal mechanisms in specifying the construction process in order to keep the exposition simple.) Evaluation of an immediate (non-upon) precondition leads to an error if it is evaluated to FALSE. Evaluation of *upon* condition is based on *wait* semantics and thus its evaluation to FALSE does not lead to an *error*, rather the *upon*condition is re-evaluated again (but in a different atomic unit of execution). Application of such semantics can be seen in the valuation function for evaluation of methods.

Booleans: TRUE, FALSE $\in \mathbb{B}$.
Operations:
and: $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
or: $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
not: $\mathbb{B} \rightarrow \mathbb{B}$

Objects:
SafeObjects: set of objects that are in safe state with respect to *upon* conditions.
error: an error state in which all the mutual exclusive accesses to objects are relinquished.
$[\![M]\!]o$: an application (non-atomic) of M over the object o(evaluation of M in o).
$[\![\overline{M}^{\alpha}]\!]o$: an *atomic* application of M over the object o with α as the unit of atomicity.

Operations:
constructor: $\Gamma \rightarrow$ *SafeObjects*
uponset: $\mathcal{M} \times \Gamma \rightarrow \mathcal{E}$
non-upon: $\mathcal{M} \rightarrow \mathcal{C}$
typeOf: $Objects \rightarrow \Gamma$
equalTypes: $\Gamma \times \Gamma \rightarrow \mathbb{B}$
equals: *uponset* $\times$ *uponset* $\rightarrow \mathbb{B}$
$(\_ \rightarrow_{\_} \square\ \_)$: $\mathbb{B} \times$ *DOM* $\times$ *DOM* $\rightarrow$ *DOM* for a previously defined domain *DOM*.

Fig. 11. Semantic Algebra

Evaluation of a method on an object has the following semantics: if a supertype client invokes the method on o, then it proceeds with just the evaluation of the method unlike when a subtype client invokes such a method. A subtype client changes the state of the object to an unsafe state (as it encounters an *upon* condition); in such a state the method M is evaluated under an atomic unit of execution α. Its evaluation of M is followed (not necessarily immediately) by changing the state of o to a safestate through *setsafe*. The atomic unit under which, such a state change occurs is important to the context in which the object is supposed to be accessed. If o is accessed by a mix of subtype and supertype clients, then evaluation of *setsafe* is carried out under the same

atomic unit (α) as specified in the valuation function. However if o is accessed by only subtype clients, then the mutual access to the object can be relinquished in an unsafe state (Section 6); therefore the atomic unit under which o could be different than the one (α) under which the invoked method is evaluated. For a scenario in which an object o is accessed only by subtype clients, the valuation function is as follows:

$E[\![o.M()]\!] = E[\![M]\!](o)$
$= \mathit{equals}(\mathit{uponset}(M, \mathit{typeOf}(o)),\ \phi) \rightarrow E[\![\overline{M}^{\alpha}]\!](o) \,\square\, (E[\![\overline{M}^{\alpha}]\!](o);\ E[\![\overline{\mathit{setsafe}}^{\beta}]\!](o))$

$E[\![o\ := \mathbf{new}\ t]\!] = \mathit{constructor}(t) \rightarrow o.$

$B[\![B]\!](o) = C[\![C]\!](o) \rightarrow \text{TRUE} \,\square\, \mathit{error}.$
$= U[\![U]\!](o) \rightarrow \text{TRUE} \,\square\, U[\![U]\!](o).$
$A[\![A]\!](o) = A[\![C]\!](o)$

$E[\![o.M()]\!] = E[\![M]\!](o)$
$= \mathit{equals}(\mathit{uponset}(M, \mathit{typeOf}(o)),\ \phi) \rightarrow E[\![\overline{M}^{\alpha}]\!](o) \,\square\, (E[\![\overline{M}^{\alpha}]\!](o);\ E[\![\overline{\mathit{setsafe}}^{\alpha}]\!](o))$

$E[\![\overline{M}^{\alpha}]\!](o) = M[\![\overline{(B,S,A)}^{\alpha}]\!](o) = \overline{B[\![B]\!](o);\ S[\![S]\!](o);\ A[\![A]\!](o)}^{\alpha}$
$= \overline{U[\![U]\!](o) \rightarrow\ (C[\![C]\!](o) \rightarrow\ (S[\![S]\!](o);\ A[\![A]\!](o)) \square\ \mathit{error})}^{\alpha} \,\square\, M[\![\overline{(B,S,A)}^{\beta}]\!](o)$
$= U[\![\overline{U}^{\alpha}]\!](o) \rightarrow (C[\![\overline{C}^{\alpha}]\!](o) \rightarrow (S[\![\overline{S}^{\alpha}]\!](o); A[\![\overline{A}^{\alpha}]\!](o)) \,\square\, \overline{\mathit{error}}^{\alpha}) \,\square\, M[\![\overline{(B,S,A)}^{\beta}]\!](o).$

Fig. 12. Valuation Functions for selected domains

5.1 Resumptions

Let the supertype and the subtype be σ and ω, respectively. Let o be an object constructed from ω. Let M be a method defined in σ, which is redefined as $\tilde{M}$ in ω.

Resumption-I : An invocation of M on o by supertype client through an alias $o_2{:}\sigma$ is in concurrence with an invocation of M on o by another supertype client through an alias $o_1{:}\sigma$. $\|$ denotes the concurrency (parallelism) among operations/evaluations and ";" denotes sequential evaluation. Without loss of generality, consider the sequential execution in which the invocation through o_1 precedes that through the o_2, which is denoted as follows:

$E[\![o_1.M()]\!] \,\|\, E[\![o_2.M()]\!] = M[\![\overline{M}^{\alpha}]\!](o_1);\ M[\![\overline{M}^{\beta}]\!](o_2)$

Let us expand $M[\![\overline{M}^{\alpha}]\!](o_1)$ as follows:

$M[\![\overline{M}^{\alpha}]\!](o_1)$
$\Rightarrow o_1 \in$ *SafeObjects*
$\Rightarrow o \in$ *SafeObjects*

$o \in$ *SafeObjects and* $o = o_2$
$\Rightarrow o_2 \in$ *SafeObjects*
$\Rightarrow$ *upon*(o_2, σ)
$\Rightarrow$ *upon* condition for M in o_2 does not block indefinitely.
$\Rightarrow E[\![(o_2).M()]\!] = C[\![\overline{C}^{\beta}]\!](o) \rightarrow (S[\![\overline{S}^{\beta}]\!](o); A[\![\overline{A}^{\beta}]\!](o)) \,\square\, \overline{error}^{\beta}$.

A change of the order of the above two operations would proceed with no blocking and follows from the above semantics.

Resumption-II : An invocation of M on o by supertype client through an alias $o_2{:}\sigma$ is concurrently being implemented by an invocation of M on o by a subtype client through an alias $o_1{:}\omega$. We are specifically interested in the sequential execution in which the invocation of subtype client precedes that of the supertype client, which is specified as follows:

$E[\![o_1.M()]\!] \,||\, E[\![o_2.M()]\!] = M[\![\overline{M}^{\alpha}]\!](o_1);\, M[\![\overline{M}^{\beta}]\!](o_2)$
$=$ *equals*(*uponset*(M, *typeOf*(o_1)), ϕ) $\rightarrow E[\![\overline{M}^{\alpha}]\!](o_1) \,\square\, (E[\![\overline{M}^{\alpha}]\!](o_1);\, E[\![\overline{setsafe}^{\alpha}]\!](o_1));$
$M[\![\overline{M}^{\beta}]\!](o_2)$

In addition,

equalTypes(*typeOf*(o_1), ω)
$\Rightarrow$ *not equals*(*uponset*(M, ω), ϕ)
$\Rightarrow M[\![\overline{M}^{\alpha}]\!](o_1) = E[\![\overline{M}^{\alpha}]\!](o_1);\, E[\![\overline{setsafe}^{\alpha}]\!](o_1)$

Moreover,

$E[\![\overline{setsafe}^{\alpha}]\!](o_1)$
$\Rightarrow o_1 \in$ *SafeObjects*
$\Rightarrow o \in$ *SafeObjects*

$o \in$ *SafeObjects and* $o = o_2$
$\Rightarrow o_2 \in$ *SafeObjects*
$\Rightarrow$ *upon*(o_2, σ)
$\Rightarrow$ *upon* condition for M in o_2 does not block indefinitely.
$\Rightarrow E[\![(o_2).M()]\!] = C[\![\overline{C}^{\beta}]\!](o) \rightarrow (S[\![\overline{S}^{\beta}]\!](o); A[\![\overline{A}^{\beta}]\!](o)) \,\square\, \overline{error}^{\beta}$.

Therefore,

$M[\![\overline{M}^{\alpha}]\!](o_1)$

$\Rightarrow E[\![(o_2).M()]\!] = C[\![\overline{C}^{\beta}]\!](o) \rightarrow (S[\![\overline{S}^{\beta}]\!](o); A[\![\overline{A}^{\beta}]\!](o)) \,\square\, \overline{error}^{\beta}$
$\Rightarrow M[\![\overline{M}^{\beta}]\!](o_2)$ does not block indefinitely on the *upon* condition.

Resumption-III : An invocation of M on o by subtype client through an alias $o_2{:}\gamma$ is concurrently being implemented by an invocation of M on o by a subtype client through an alias $o_1{:}\omega$. such a resumption could be proven to be non-blocking in a line similar to that in *Resumption-II*.

We prove Lemma 1 formally using denotational semantics using resumption semantics proposed above as follows.

Lemma 2. *A supertype client will never block on an upon condition in an object of a relaxed behavioral subtype (of the supertype).*

Proof. Let o be an object constructed from a type ω involving an *upon* condition.

Case-I: After its construction, o has not been accessed by any client before the supertype client attempts to access it;
$o \in$ *SafeObjects*$\Rightarrow$ *upon* condition for a method M in o does not block indefinitely.

Case-II: Only supertype clients are accessing the object. The lemma in this scenario follows from *Resumption-I*.

Case-III: A mix of supertype and subtype clients are accessing the object. The lemma in this scenario follows from *Resumptions-II and III*. QED

6 Nesting and Deadlocks

This section discusses the relaxed behavioral subtyping model presented in the previous section in the face of deadlocks, and presents runtime support, implemented in the context of Eiffel, required to avoid these. The presentation is made in an incremental manner with respect to the traditional subtyping case $<:_T$, and starts off with background information on the Eiffel concurrency model.

6.1 Background: Concurrency in Eiffel

Eiffel, just like Spec#, integrate assertions into the language. Eiffel is furthermore in the process of being extended with inherent support for concurrency, in the aim of providing concise concepts for safe multi-threaded programming. Currently, these concepts are put to work with a pre compiler, which generates synchronization code from contracts (see Section 7.1).

As already mentioned, the basic model [26, 25] for concurrent programming in Eiffel, dubbed SCOOP (Simple Concurrent Object Oriented Programming [26, 2]) originally introduces a single keyword for guiding synchronization explicitly — `separate` — which is used to tag variables. Roughly, every object is associated with a (single)

conceptual processor, which is the only one to execute the methods of that object. An entity which is `separate` denotes an object which is *potentially* under control of another processor. Method calls inherently trigger synchronization, i.e., before the execution of a method containing separate formal arguments, the current processor has to acquire exclusive locks on all the separate objects passed as actual parameters, more precisely on their processors, since these execute a single method at a time. This forms the basic synchronization scheme. As a consequence, all separate objects used by a method must be arguments of that method.

Concurrency is increased by allowing methods without return values on separate objects to perform asynchronously, and only synchronizing upon methods with return values. Several extensions have been introduced recently, to relax or extend the model [2, 27], or to provide seemingly atomic execution of methods through transactional support [36, 9], but we focus on the above model in this context.

6.2 Preliminary: Nesting and Inherent Locking

When invoking an object through a method encompassing a *correctness* condition, the condition is "passed on" to the client site, i.e., the client performing the invocation is responsible that the condition is ensured. In a nested invocation scenario, this means that such correctness conditions can accumulate inversely to the nesting order of invocations.

If *wait* conditions are used as synchronization barriers for methods, then corresponding conditions can similarly propagate. Take a method m_1 with a wait condition B_1 that is invoked within another method m_2. The invocation to m_2 may be preceded by an `if` (B_1) `...` statement, or possibly m_2's precondition B_2 will include B_1; in both cases, if one can infer parts B of B_1 ($B \subseteq B_1$) at the moment of the invocation, such that it is sufficient to verify $B_1 \backslash B$ (which might even be empty – $\emptyset$). Since we adopt the Eiffel model of concurrency (see Section 7), the `if` statement (if any is needed) which represents a guard gets merged with B_2. Consider in the following the example of a buffer invoked through a variable of type *BUFFER* from an auxiliary class defining a method for the atomic addition of a set of elements:

```
feature put_n(objects: ARRAY[G]; buffer: separate BUFFER[G]) is
require
  objects.length <= buffer.max - buffer.count
do
  [...]
end -- put_n
```

The precondition of *put_n()* needs to reflect the cumulated precondition of the individual *put()*s if *put_n()* is to execute atomically on *b*.

This essentially yields two scenarios for invocations to shared data structures such as our *BUFFER* example:

Nested synchronization: If mutual exclusion is desired, method invocations to a concurrent data structure are surrounded by (possibly implicit) lock acquire and release operations. We will refer to these scenarios as *nesting* meaning that synchronization is nested, because the reasoning about mutual exclusion etc. propagates.

Non-nested synchronization: If mutual exclusion is not required, no locks need to be acquired/released before and after an invocation respectively, at least *not explicitly*. We will refer to such cases as *non-nesting* cases, since locks are not inherited to encapsulating methods.

Ultimately, any program consists in the execution of blocks containing (nested) synchronization and such containing no synchronization. The difficulties in reasoning arise at the borders of the two. Reasoning about deadlock-freedom in the presence of individual calls without any annotations or syntax whatsoever for synchronization is not tractable and has not been addressed in literature. We focus in the following on the former case, and subsequently come back to the latter case in the case of *implicit* synchronization of such individual calls. The challenge arising in the former case consists in ensuring that any added wait condition is considered if an object is referred to and invoked through a variable of a a super-type. To that end, the generation of synchronization barriers and guards must foresee that an objects dynamic type might introduce additional wait conditions, and thus cater for this case. In the above example, any individual call to `put()` will evaluate the additional wait condition if `b` refers to an object of type `CONG_CTRL_BUFFER`. Section 7 elaborates more on this.

6.3 Separating Clients

In [7], Dhara and Leavens present a model of *weak* behavioral subtyping (see Section 9), which founds on the assumption that there is no aliasing across subtypes. In other terms, all references to a given instance of a buffer subtype e.g. `LOCKABLE_BUFFER` occur through variables b_i of the exact same type – all b_i are either of type `BUFFER` or of type `CONG_CTRL_BUFFER`.

It is easy to see that in such scenarios no deadlocks are introduced in our case of $<:_S$. If an object of a type $T_2 <:_S T_1$ is referenced by variables of type T_1 (only), it behaves exactly like an instance of T_1 given that the object starts in its safe state and that this mode can only be switched by methods added by T_2. If all variables to such an object are of type T_2, the same reasoning about deadlock applies as for any behavioral subtyping scenario. thus boils down to the traditional problem.

In the following scenarios where $T_2 <:_S T_1$, client objects referring to instances of a type T_2 through variables of type T_1 are termed *original clients*, and objects detaining references of type T_2 to objects of type T_2 as *subtype clients*.

6.4 Nested Synchronization

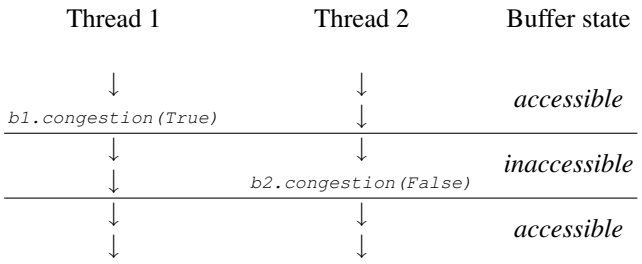
A premise for well-behaving programs is that wait conditions added by subtypes do become active in the first place with original clients, despite the presence of late binding and polymorphism. As mentioned above, this is handled by our implementation by generating synchronization code for each method of each class requiring such synchronization (in the case of Eiffel meaning involving `separate` objects), and executing that code on the invokee side (see next section). The interesting case arises now with objects being accessed by both original and subtype clients. We distinguish in the following two cases which allow to narrow down the additional required reasoning about concurrency, and especially, deadlocks.

Reverting to safe states in subtype clients. The first case consists in assuming that all subtype clients of an object revert the object to the safe state at the end of a series of calls on that object within mutual exclusion, whether implicitly or explicitly, before releasing the lock on that object. In this case,no additional deadlocks are trivially introduced: no (original) client finds an object in an extended state, i.e., a state in which added fields are of relevance. Thus, no extended wait conditions can be in the way. In other terms, a client which accesses a same buffer of type `LOCKABLE_BUFFER` as the client in Section 6.2 can only make use of `congestion()` if it subsequently performs an inverse `congestion()` before releasing the buffer:

```
CONG_CTRL_BUFFER[G]) is require not buffer.congestioned do
buffer.congestion(True) wait(1000000) buffer.congestion(False) end -- de_congest
```

The object referred to by `b` could not be released without the second call to `congestion()`, as no client referring to the same object through a variable of type `BUFFER` could use the object. More precisely, such a client can lock the object prior to the intended usage, and find the extended precondition of `congestion()` to not hold.

Unlocking in extended states. The above is a strong restriction for many cases. Why wouldn't it be possible to split the two calls to `congestion()`? They could be executed by different threads on a same buffer referred to by each through respective variables b_1 and b_2, as follows, where b_i refers to a buffer:



This would lead to a rendez-vous style interaction possibly with an inverse execution order, with each method being guarded by the respective precondition of `CONG_CTRL_BUFFER`. Original clients could be running concurrently, without *necessarily* introducing deadlocks.

We discuss in the following how to relax the above restriction such as to allow such executions, i.e., to allow subtype clients to "release" objects in an extended state in the presence of original clients.

The idea is rather intuitive. It essentially means that for any client using an object through a variable of type T in a synchronized setting, one has to ensure that any object accessed through that variable passing the synchronization point is actually in the safe state associated with type TThis adds no additional requirements for subtype hierarchies following traditional behavioral subtype conformance $<:_T$. In the context of $<:_S$, this requires the addition of (a) wait condition(s) on the corresponding field(s) which are enabled only by the safe state, together with the lock acquire. As a consequence, an original client never gets to execute in an extended state, even if it was possible to perform the corresponding methods. Section 7.2 presents different ways of implementing

such a wait condition for the safe state, including automatic and programmer-assisted schemes.

For the deadlock reasoning, the above way of proceeding retains modularity, by clearly separating the reasoning into (1) original clients (with respect to the semantics of the original type) as traditionally the case and (2) confined to subtype clients *among themselves*. Intuitively, one can think of such objects being *shared* by original and subtype *programs* along the lines of weak behavioral subtyping, with transitions occurring only in states which are safe for either side to execute in.

6.5 Non-nested Synchronization

Suppose an invocation to a shared data structure takes place in mutual exclusion with respect to other calls to that data structure, but is not within a nested synchronization block. The solution outlined above in the case of nested synchronization can be applied to this case as well. That is, the use of mixed clients (original and subtype clients) requires any call to such an object coming from an original client to be inherently guarded by a wait condition for the corresponding safe state. In the case of nested synchronization this wait condition was already executed before-hand, and thus is trivially satisfied.

Note that the less runtime-intrusive but for programmers more restrictive approach of requiring any subtype client to revert to the original state — given that we are considering a single invocation in this case — would mean that there would be no means to even enable an extended state.

7 Implementation

In this section, we illustrate the concept of strengthening preconditions through the SCOOP model of Eiffel. A background on the SCOOP precompiler relevant to the context of strengthening of preconditions is presented.

7.1 Background: SCOOP Precompiler

A SCOOP precompiler (*scoop2scoopli* [35]) transforms a SCOOP program to an Eiffel program based on calls to a SCOOP library. The SCOOP library (*scoopli* [35]) handles the concurrency semantics as specified through the `separate` keyword. In the transformed code, each object is associated with a processor, which in turn is managed by a scheduler. The scheduler manages acquiring locks on the receiver of a method call and all the separate arguments of the method. An object is locked by acquiring a lock on its processor. For the call to proceed, precondition of the method has to be satisfied, which in turn requires that the locked receiver and other locked objects are in a state satisfying all the clauses of the precondition involving them. All these preconditions are propagated up to the caller routine of the method. The scheduler evaluates these clauses until they hold, before locking the receiver object on behalf of the caller. In other words, the caller waits for these clauses of the precondition to hold before being able to acquire the lock.

The transformed program waits until the precondition of the method that accesses the synchronized (through `separate`) object holds and then locks the object. Such a *wait* semantic is very useful for the implementation of the strengthened preconditions in SCOOP model. The precompilation process on SCOOP-based programs, transforms the programs in order to add such semantics whenever the intention of synchronized access is specified through `separate` annotation.

The precompilation makes use of the *agent* mechanism of Eiffel (a substitute for higher-order functions which is similar to *delegates* in C#) and the SCOOP library. The SCOOP precompiler transforms

- Each class that refers to one or more instances as separate instances; such classes are called as *client classes*.
- Each class whose instance is referred to as a separate instance in a client class; such classes are called as *server class*.

Client classes. Pre-compilation of a client class transforms the class in place, and generates a proxy class - *SS_classname* (*SS* stands for *SCOOP_SEPARATE*). The proxy class refers to an instance of the transformed client class as its implementation through the field *implementation_*. A client class declares separate instances of server classes as an argument to a method (separate-by-argument). An example is: `send(buf: separate BUFFER[INTEGER],...)` in *Sender* client (Listing 14). Translation of every such method leads to generation of two new methods, each one is used to define an agent. The arguments of the new methods are identical to those of the client method.

One of the new methods - `send_sss_er()` (*sss* denotes *scoop_separate_sender*, *er* denotes *enclosing_routine*) is a wrapper method with almost identical (identical for our purpose) definition and functionality as that of the client method in translation - *send* (Listing 16). The second new method - *send_sss_wait_condition* (Listing 17) evaluates those clauses in the precondition of the client method that uses a separate object and returns the result of the evaluation as a boolean value. The SCOOP scheduler treats it as a wait condition. The method call on the server object inside the client method - `buf.get()` is replaced by an invocation to the SCOOP library routine *separate_execute_routine*`()` with the two agents as parameters (Listing 15). *separate_execute_routine*`()` passes the two agents to the SCOOP scheduler for execution. Upon acquiring all the locks, the scheduler calls the routine on the agent implementing the wait condition. If the routine returns true, the processor invokes a call on the agent of the method. If any of the wait conditions do not hold, all locks are relinquished, and the method is rescheduled for a later retrial.

Server classes. For the server classes, a proxy class is generated. Any call on a server object is routed through an instance of its proxy to the SCOOP library and the original class. The proxy uses agents for each call on a separate object and passes these agents to the library routines. *scoop_asynchronous_execute*`()` and *scoop_synchronous_execute*`()` are the library routines that enqueue the call to the processor of the specific object. The enqueued calls are scheduled for execution later by the SCOOP scheduler.

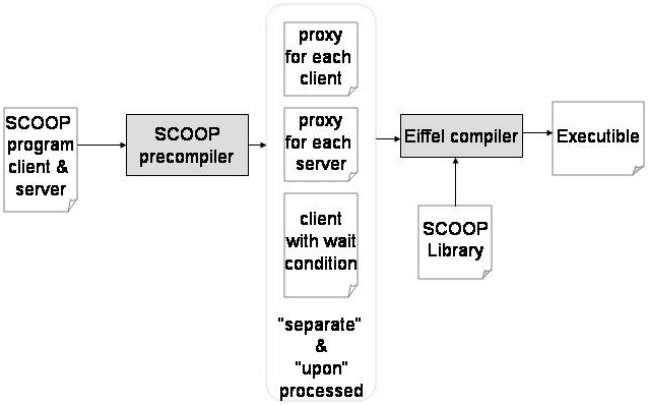


Fig. 13. Working of the SCOOP precompiler

```
class Sender
feature
  [...]
send(buf: separate BUFFER [INTEGER]) is
require
  buf /= void
do
  [...]
  ... := buf.get
  [...]
end
end
```

Fig. 14. The client `Sender` for `BUFFER`

```
send(buf: SS_BUFFER [INTEGER_32]) is
require
  buf /= void
do
  [...]

  separate_execute_routine(..., agent
        send_sss_er(buf), agent
  send_sss_wait_condition(buf),...)

  [...]
end
```

Fig. 15. Translated `send()`

7.2 Handling Strengthened Preconditions

A first extension of the SCOOP precompiler consists in handling these `upon` clauses. Handling of `upon` clauses comprises of (1) its verification and (2) replacement of `upon` keyword as the Eiffel compiler is not aware of.

Verification of the Safe State To honor the restrictions outlined in the previous section, it is important that an object (which is separate) is only locked by a client accessing that object through a variable of a given type T if the object is in its default state with respect to type T. Any method locking a set of separate objects is thus added a wait condition to ensure that all these objects are in their respective default states.

In our implementation such a wait condition contains a call to a method `is_safe()` which returns a boolean value and thus represents a predicate. This method is defined as part of the `STRENGTHENABLE` class (see Listing 19), which any class making use of `upon` clauses implements. Since recursive subclassing with `upon` clauses is possible, the `is_safe()` method takes as argument a type *name* (given the absence of meta-classes in Eiffel). Whenever an object of an extended subtype according to the model

```
send_sss_er (buf: SS_BUFFER [
     INTEGER_32]) is
require
  buf /= void
do
  [...]

  ... := buf.get(Current, el)

  [...]
end
```

Fig. 16. Generated Wrapper for `send`

```
send_sss_wait_condition (buf: SS_BUFFER [
     INTEGER_32]) : BOOLEAN is
local def: STRENGTHENABLE
do
  def := buf
  if (def /= void) then
    Result := True and then (def.is_safe(BUFFER))
      else
        Result := True
  end
end
```

Fig. 17. Generated Wait Condition for `send()`

presented in this paper appears as argument to a synchronization barrier, the barrier is added a wait condition involving a call to the above-mentioned method, passing as argument the name of the static type through which the object is being accessed.

The programmer can easily implement this method herself by means of a `switch -case` statement with the cases being superclasses of the considered class in the class hierarchy. This implementation can easily reuse the implementations of this method in super-classes. A class that strengthens a precondition through `upon` inherits `STRENGTHENABLE` and redefines the `is_safe()` method.

In the case of omission of inheritance of `STRENGTHENABLE` from a class `C` making use of `upon` clauses, or in the case the programmer does not want to be troubled, `STRENGTHENABLE` is automatically added by the precompiler to the list of parent classes of `C`, and an implementation of `is_safe()` is automatically generated. In this case, the method verifies the *disjunction of all* `upon` *clauses* in conjunction with calls to the `is_safe()` methods of any super-classes which are using `upon` clauses.

For `CONG_CTRL_BUFFER` (Listing 9) that defines upon conditions for each `put()` and `get()`, the precompiler generates `SS_CONG_CTRL_BUFFER` (Listing 18). Anytime an *upon* keyword is found in a method (e.g. `put()`) in the class to be transformed, the precompiler checks if the class is a descendant of `STRENGTHENABLE`. If not, the generated class is defined to `inherit STRENGTHENABLE` and the programmer is notified to implement the `is_safe()`. The `is_safe()` is conjoined with the predicate evaluation statement in the method for wait condition generated as part of standard SCOOP precompilation (Section 7.1). The replacement of the upon precondition is described in the following section.

Processing of *upon* Precompilation does not remove the upon preconditions from a class after the corresponding wait condition is generated as described in the previous section; but it replaces the `upon` keyword which is not an Eiffel keyword. It is desirable to leave the precondition of a method in a class remain as that would facilitate processes as debugging and program maintenance. Since strengthening of preconditions is not supported in Eiffel language, the condition specified through `upon` needs to be passed in a manner that the compiler would not complain as illegal.

It is well known that a precondition involving a call to a method that evaluates a predicate can be strengthened (see Section 8.3) in a subclass by redefining the method

```
class SS_CONG_CTRL_BUFFER[G]
  inherit SS_BUFFER[G], STRENGTHENABLE
feature [...]

-- redefine put and get

redefine is_safe (superclass: String) :
     Boolean is
do
  if (superclass = BUFFER) then
    Result := True and then (not
         congestioned)
    else Result := True
  end
end

redefine upon_one: BOOLEAN is -- for put
do
  Result := not congestioned
end

redefine upon_two: BOOLEAN is -- for get
do
  Result := not congestioned
end
end
```

Fig. 18. Separate congestion control buffer after Precompilation

```
class STRENGTHENABLE
feature [...]

is_safe(superclass: String):
     BOOLEAN is
do
  Result := True
end

end -- STRENGTHENABLE
```

Fig. 19. STRENGTHENABLE class

and making it evaluate a stronger predicate. Since this paper characterizes a certain class of conditions (upon conditions) can be used to safely strengthen preconditions, the modified SCOOP precompiler processes the *upon* precondition of a method `m()` in class `C` as follows:

- it identifies the super-class `A` of `C` that is the first such class (earliest ancestor of `C`) in the hierarchy to define `m()`;
- defines a method `upon_x()`, where `x` is a unique 'id' for the method `m()` such that it returns TRUE; the id `x` is also used for all those methods that redefine `m()` later (in this case `upon_x()`) in `A` that returns TRUE
- modifies the precondition of `m()` by conjoining `upon_x()` with the existing precondition of `m()`
- redefines method `upon_x()` in every sub-class of `A` that locally strengthens the precondition of `m()` through an `upon` condition. The redefinition of `upon_x()` involves evaluation of the local `upon` condition.

Since the upon conditions are defined on only the internal parameters of the class, no argument needs to be passed to such methods. However using method invocations in a precondition is an un-safe way of implementing preconditions, unless implemented carefully. It is unsafe, because it may lead to breaking of contracts at the subclass level. The method maybe overridden "incorrectly" (as part of the software evolution process), which might strengthen the precondition in the subclass with conditions that are not eventual. One correct way of overriding such a method is what we are prescribing in this paper. In our implementation, we are allowing method invocation to be used to

strengthen preconditions and it is safe, because the method is redefined to strengthen the precondition through "wait" conditions and no other conditions that would make it unsafe.

8 Discussion

This section discusses different issues related to the previously introduced concepts, such as how they relate to locks.

8.1 Strengthening?

One may argue that "strengthening" only occurs seemingly, more precisely, that the conjunctions we allow for preconditions is *implicitly* a disjunction: by requiring that the conditions conjoined in subtypes with preconditions of the respective methods of the super-type have *default* values for the involved arguments which satisfy the conjoined condition for all instances of the super-type, one really has a disjunction with a stronger condition, which moreover leads to retaining the initial condition.

As an example, a strengthening resulting from a conjunction of the form

$$\downarrow C_1 \wedge \diamond C_2 \tag{9}$$

where

$$\diamond C_2 = (F \in \mathcal{R}) \tag{10}$$

for an added fled F with default value $\nabla F \in \mathcal{R}$ ($\mathcal{R}$ range of default value) yields in fact

$$(\downarrow C_1 \wedge F \in \nabla F) \vee (\downarrow C_1 \wedge F \in \mathcal{R} \backslash \{\nabla F\}) \tag{11}$$

which can be satisfied trivially by

$$\downarrow C_1, \tag{12}$$

assuming this default value is eventually ensured. From the perspective of the sub*type* and its specification however, which represents the adequate point of view, the precondition is a priori strengthened. The reasoning steps from Expression 9 to Expression 12 require further specifications and axioms such as the default value introduced by (10).

8.2 Explicit State Variables versus Locks

One can attempt to achieve the same functionality as the features added by `LOCKABLE_BUFFER` by applying external locks on a conventional `BUFFER` – with explicit lock acquire and release. Reasoning about concurrency through such locks is for instance advocated by

the recent seminal work on JML [32] in the presence of multithreading. Such an approach would then remove the need for an *intrinsic* extension of the `BUFFER` class.

As argued however in Section 2 already, the application of external locks makes a state machine-like extension of concurrent data structures harder. Moreover, applying external locks to objects can lead to a coarse grained concurrency control model; locking an entire data structure can be overkill in many more elaborate scenarios.

8.3 Use of Methods in Strengthening Preconditions

Most frameworks for behavioral subtyping currently implicitly offer a workaround for the impossibility of strengthening preconditions. By defining a *precondition method* for each method, where the former simply implements the precondition of the latter method and can thus be overridden in a subclass by strengthening it, a similar effect can be achieved as with our extension. Consider the following case of a class `C` with a method `m()`:

```
class C
feature
  m is
  require
    m_pre
  do
    ...
  end -- m

  m_pre is ...
```

Overriding `m_pre()` in a subclass `D` of `C` does the trick. However, this is far from being safe, and in particular puts more burden on the programmer by concealing the important details of `m_pre()`. This illustrates the hardness of taming complexity in behavioral subtyping, as a "pure" approach would require `m_pre()` to be defined with contracts as well which might contradict then a "strengthening" of `m_pre()` in a subclass `D`.

In contrast, our approach lays the strengthening of preconditions open, and provides a disciplined approach to reasoning about such scenarios.

9 Related Work

This section overviews closely related work, positioning it with respect to our present work.

9.1 Axiomatic Semantics and Concurrency

Axiomatic semantics [13], also referred to as Hoare-logic, were initially introduced without concurrency in mind.

One of the seminal efforts to tackle concurrency in the context of axiomatic semantics is the well-known Owicki-Gries approach [29], which provides a framework for determining under what conditions two blocks of code can be executed in parallel without producing data-races. Owicki-Gries basically evaluates for every single instruction

in a first code block whether it conflicts with either of the instructions of the second block. [29] thus requires $O(n + m)$ single verifications for two code blocks of n and m elementary instructions respectively (without mentioning the possibility of nested method calls in the context of object-oriented programming). In this paper, we consider that methods run in mutual exclusion, but only because it simplifies presentation. If two methods can be proven to not conflict along the lines of [29], they may also be executed simultaneously in our case.

Lipton's [19] early work aims at identifying atomic regions in programs, i.e., regions which can be considered to execute atomically. Such atomic regions then can typically reduce complexity in Owicki-Gries. Without going into detail, [19] classifies individual instructions by the possibility of preceding or succeeding (or both) them by instructions of other threads. This method is particularly interesting when explicitly using locks (acquires and releases) inside the code, as those remove many possible interleavings.

The *rely-guarantee* approach [16] has been introduced as a computationally less expensive alternative to Owicki-Gries, which is additionally compositional. In the latter method, any change of one of the components may namely affect the proof of all components. The approach of [16] consists in augmenting Hoare-triples with *rely* and *guar* predicates. A corresponding code block then is specified as executing as soon as its precondition holds, with only *rely* being subsequently required to remain valid, and *guar* being ensured, before the end of the code at which only the (full) postcondition must hold. The resulting slightly more complicated rule for parallel composition however is compositional, and has lower complexity, the downside being that it potentially requires more input from the programmer.

The work of [33] has in much inspired the approach presented in this paper. Though not explicitly targeting a subtyping/inheritance scenario, the incremental nature of program development and proving, including addition of state variables, proposed has much in common with such a subtyping setting. [33] *strengthening*, as they focuses on extending finite automata for which the possible orderings of states are implicitly defined by transition functions. tackle the proving of deadlock-freedom in a more general manner, i.e., by proving pairwise deadlock-freedom between any two state machines. Without considering nested calls in an object setting, more relaxed semantics are even possible. In contrast, we consider here a single buffer (type), and focus on the use by respective clients.

Various other authors have considered stepwise and incremental development of systems specified with axioms. Gribomont [12] for instance focuses on incremental development in the sense of refinement, where existing specifications and automata are *modified* rather than augmented through an inheritance mechanism and included into polymorphism.

Discouraged by the hardness of specifying axioms in the presence of multithreading, given the intertwining of application logic and synchronization requirements, many hopes were put in *temporal logic* [] to specify synchronization patterns in isolation from methods. Temporal logic-based approaches tend to suffer from complexity, as potentially infinite histories of events must be tracked.

9.2 Behavioral Subtyping and Concurrency

Alike the seminal work on axiomatic semantics outlined above, most work in behavioral frameworks for object-oriented programming taking into account multithreading aims at increasing parallelism by allowing for simultaneous execution of code blocks, or methods, respectively. The goal consists then in proving non-interference, and a way to achieve that (seeming) atomic execution of code blocks. The latest work along those lines is [32], which contains a broad and detailed survey of related approaches. the following.

[15] extends Spec#, a variant of C#, with specifications, to deal with multithreaded programs. As in Eiffel, specifications are part of the language. [15] emphasizes two keys concepts for multithreaded programs, namely (1) aliasing and (2) locking, which are dealt with through keywords `pack`/`unpack` and `acquire`/`release` respectively. The authors' main focus is on the preservation of invariants in the presence of multithreading.

[32], alike [15], models locks which are implementation-related in JML (specifications). An extensive syntax is added to define locks, describe which objects they relate to and how, etc. The results described derive from a collection of techniques applied, inspired, and adapted from various sources including [11, 37, 31]. Among the syntax introduced is also a **`when`** clause, which, alike a guard or the eventual conditions described in this paper, can be used to describe blocking behavior of a method. Their exact relationship to preconditions is however not described. With respect to [15], the techniques described in [32] seem more exhaustive, but sill more flexible, as they allow for adaptation of existing Java programs, and do not prescribe strong constraints (e.g., locks can be shared).

The SCOOP model of concurrency proposed in conjunction with the Eiffel programming language has strongly motivated the work presented in this paper, in particular because preconditions are monitored at runtime in Eiffel, and in SCOOP are used as synchronization mechanism. [28] dissects both pre- and postconditions in the presence of multithreading, separating between eventual and immediate preconditions ("wait semantics" and "correctness semantics" respectively). However, the authors do not proceed to further distinguish between method arguments and fields of the receiver object, which is the key to strengthening preconditions.

Though not addressing concurrency (explicitly, see Section 1), the work of [7] is worth mentioning at this point. The authors introduce a notion of *weak* behavioral subtyping, which allows types with mutable objects (mutable types) to have subtypes with immutable objects (non-mutable types), unlike any of the original definitions of [20]. The consequence of this, is that objects may not be referenced through variables of distinct supertypes. To that end, a static program analysis is proposed. As pointed out by the authors of [7], the collection hierarchy considered in [5] conforms to their weak notion of subtyping.

10 Conclusions

Behavioral subtyping lays the foundation for modular reasoning about programs, and thus represents a cornerstone towards provably correct software. It has however often

been argued that the traditional constructs of behavioral subtyping are overly constraining, and preclude useful scenarios.

In this paper we exhibit such a simple yet representative scenario, where preconditions need to be *strengthened*, going against the popular confines of behavioral subtyping. We discuss the role of multi-threading therein, and introduce precise guidelines following which such a subtyping does not necessarily hamper safety of programs, and modular reasoning about correctness as well as deadlock-freedom is still supported. We present the implementation of this relaxed model in the context of the Eiffel programming language.

Acknowledgment We thank Jean Privat, for his discussions and comments on the paper.

References

1. S. Ajmani, B. Liskov, and L. Shrira. Modular Software Upgrades for Distributed Systems. In *19th European Conference on Object-Oriented Programming (ECOOP'06)*, pages 452–476, 2006.
2. V. Arslan, P. Eugster, P. Nienaltowki, and S. Vaucouleur. *Dependable Systems: Software, Computing, Networks*, chapter SCOOP – Concurrency Made Easy. Number 4028 in Lecture Notes in Computer Science. Springer, 2006.
3. J.-P. Briot. Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. In *3rd European Conference on Object-Oriented Programming (ECOOP'89)*, pages 109–129, 1989.
4. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Rustan, M. Leino, and E. Poll. An Overview of JML Tools and Applications. *Electronic Notes in Theoretical Computer Science*, 80, 2003.
5. W. Cook, W. Hill, and P. Canning. Inheritance Is Not Subtyping. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL '90)*, pages 125–135, 1990.
6. L. Crnogorac, A. S. Rao, and K. Ramamohanarao. Classifying Inheritance Mechanisms in Concurrent Object Oriented Programming. In *12th European Conference on Object-Oriented Programming (ECOOP'98)*, pages 571–600, 1998.
7. K. Dhara and G. Leavens. Weak Behavioral Subtyping for Types with Mutable Objects. *Electronic Notes on Theoretical Computer Science*, 1, 1995.
8. Eiffel Software. *Building bug-free O-O software: An introduction to Design by Contract™* `http://archive.eiffel.com/doc/manuals/technology/contract/`.
9. P. Eugster and S. Vaucouleur. Composing Atomic Features. *Science of Computer Programming*, 63:130–146, 2006.
10. M. Fähndrich and K. Leino. Declaring and Checking non-null Types in an Object-Oriented Language. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '03)*, pages 302–312, 2003.
11. C. Flanagan and S. Freund. Atomizer: a Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL '04)*, pages 256–267, 2004.
12. E. Gribomont. Development of Concurrent Systems by Incremental Transformations. In *3rd European Symposium on Programming (ESOP '90)*, pages 161–176, 1990.

13. C. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–585, 1969.
14. B. Hull, K. Jamieson, and H. Balakrishnan. Mitigating Congestion in Wireless Sensor Networks. In *Proceedings of the 2nd ACM International Conference on Embedded Networked Sensor Systems (SenSys 2004)*, pages 134–147, 2004.
15. B. Jacobs, K. Leino, and W. Schulte. Verification of Multithreaded Object-Oriented Programs with Invariants. In *ACM Workshop on Specification and Verification of Component Based Systems*, 2004.
16. C. B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.
17. G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, 2005.
18. K. Leino and P. Müller. Object Invariants in Dynamic Contexts. In *18th European Conference on Object-Oriented Programming (ECOOP '04)*, 2004.
19. R. Lipton. Reduction: a Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717–721, 1975.
20. B. Liskov and J. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
21. M. Barnett and K.R.M. Leino and Wolfram Schulte. *CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, chapter The Spec# Programming System: An Overview. Springer, 2004.
22. S. Matsuoka and A. Yonezawa. Analysis of Inheritance Anomaly in Object-oriented Concurrent Programming Languages. In *Research Directions in Concurrent Object-oriented Programming*, pages 107–150. MIT Press, 1993.
23. B. Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, 1992.
24. B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, 1992.
25. B. Meyer. *Object-Oriented Software Construction*, chapter Concurrency, Distribution, Client-Server and the Internet. Prentice-Hall, 2nd edition, 1998.
26. B. Meyer. Systematic Concurrent Object-Oriented Programming. *Communications of the ACM*, 36(9):56–80, 2002.
27. P. Nienaltowski. Flexible Locking in SCOOP. In *First International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, 2006.
28. P. Nienaltowski and B. Meyer. Contracts for Concurrency. In *First International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, 2006.
29. S. Owicki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–285, 1976.
30. S. Rangwala, R. Gummadi, R. Govindan, and K. Psounis. Interference-aware Fair Rate Control in Wireless Sensor Networks. In *Proceedings of the ACM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2006)*, pages 63–74, 2006.
31. Robby, E. Rodríguez, M. Dwyer, and J. Hatcliff. Checking JML Specifications using an Extensible Software Model Checking Framework. *International Journal on Software Tools for Technologly Transfer*, 8(3):280–299, 2006.
32. E. Rodríguez, M. Dwyer, C.Flanagan, J.Hatcliff, G. Leavens, and Robby. Extending JML for Modular Specification and Verification of Multi-threaded Programs. In *19th European Conference on Object-Oriented Programming (ECOOP '05)*, pages 551–576, 2005.
33. S. Kleuker. Incremental Development of Deadlock-Free Communicating Systems. In *3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS '97)*, pages 306–321, 1997.
34. D. A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

35. SCOOP Precompiler and Library. *SCOOP Precompiler and Library* `http://se.ethz.ch/research/scoop/index.html`.
36. S. Vaucouleur and P. Eugster. Atomic Features. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL) at the 20th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, 2005.
37. L. Wang and S. D. Stoller. Runtime Analysis of Atomicity for Multithreaded Programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, 2006.