Reasoning about Asynchronous Method Calls and Inheritance

Johan Dovland, Einar Broch Johnsen, and Olaf Owe
Department of Informatics, University of Oslo

Abstract

This paper considers the problem of reusing synchronization constraints for concurrent objects with asynchronous method calls. Our approach extends the Creol language with a specialized composition operator expressing synchronized merge. The use of synchronized merge allows synchronization classes to be added and combined with general purpose classes by means of multiple inheritance. The paper presents proof rules for synchronized merge and several examples.

1 Introduction

Distributed systems have become increasingly important in todays world, and are used in many applications, both over the Internet and local networks. Object orientation and component design are recommended for this setting by e.g. RM-ODP [18] and they are part of the philosophy of .Net. However, the integration of concurrency and asynchronous communication with object orientation is unsettled. The present paper is part of an approach combining the essential object-oriented principles with asynchronously communicating concurrent objects. In particular, asynchronous method calls and processor release points are introduced in the Creol language [21] to avoid waiting for replies to remote method calls, extending the notion of future variable [28]. In this paper we focus on language mechanisms allowing reuse of code and partial correctness reasoning, and thereby solving problems related to inheritance anomalies. The mechanisms proposed are integrated in Creol and supported by its run-time system [22].

Inheritance is a powerful structuring mechanism for code reuse. Class extension and redefinition aid in both development and understanding of code. Calling superclass methods in a subclass method enables reuse in redefined methods, making the relationship between the methods explicit. This paper focuses on reasoning control combined with *multiple inheritance* and processor release points. It is possible to inherit only a subset of the attributes and methods of a superclass, but this requires considerable work establishing invariants for parts of the superclass that appear desirable for inheritance, either anticipating future needs or while designing subclasses. Reasoning considerations therefore suggest that all attributes and methods of a superclass are inherited, but method redefinition may violate the semantic requirements of an interface. The *encapsulation principle* for class inheritance states that it should suffice to work at the subclass level to ensure that the subclass is well-behaved when inheriting from a superclass: Code design and new proof obligations should occur in the subclass only. Inheritance anomalies [24, 25] are situations breaking this encapsulation principle, in which reuse requires redefinition. In this paper, we consider a language construct for *synchronized merge* combined with multiple inheritance and its use to deal with some inheritance anomalies.

In: C. Rong (Ed.): *Proc. Norwegian Informatics Conference (NIK 2004)*, p. 213–224.
Tapir Forlag, Nov. 2004.

2 The Programming Language

We present a subset of the Creol language [21,22] sufficient for the discussion of the paper, including the constructs for inheritance and synchronized merge used in the examples. We distinguish two kinds of variables; data variables are typed by an abstract data type and object variables are typed by an interface. A class may implement several interfaces and an interface may be implemented by several classes. If a class C implements an interface I, object instances of C may have type I.

Classes organize attributes (variables local to objects) and method declarations, and may have value and object parameters as in Simula. We consider multiple inheritance where all attributes and methods of a superclass are inherited by the subclass, and where superclass methods may be redefined. To distinguish attributes or methods declarations with the same name inherited from different classes, these may be subscripted with the name of the class in which they are declared. An attribute or method x declared in a class C is uniquely named by x_C (assuming no overloading of names within a class).

Class inheritance is declared by a keyword **inherits** taking as argument an *inheritance list* of class names $C(\mathrm{E})$, where E provides actual class parameters. A method is defined *above* a class C if it is declared in C or in at least one of the classes inherited by C. When a method is invoked in an object o of class C, a method body is identified in the inheritance graph and bound to the call. To simplify the exposition, the method call is bound to the first method definition above C in the inheritance graph, in a left-first depth-first order.

The encapsulation provided by interfaces suggests that external calls to an object of class C are *virtually* bound to the closest method definition above C. However, the object may internally invoke methods of its superclasses. In the setting of multiple inheritance and overloading, methods defined in a superclass may be accessed in the subclass using subscripted references. If C' is a superclass of C, we introduce the syntax $m_{C'}(In; Out)$ for (synchronous) local invocation of a method above C. As the binding of such calls may be done without knowing the actual class of the object, they are called *static*.

Objects are dynamically created instances of classes. Object attributes are encapsulated and can only be accessed via the object's methods. In addition to object attributes, each method instance has its own local variables. The language provides a mechanism for asynchronous method invocation, where calls are explicitly labeled and computation continues without waiting for the reply to the call. We can later ask for the reply by referring to its (unique) label value [21, 22]. Method instances may be temporarily *suspended* by processor release points in method code. We assume a common type *Data* of basic data values, such as the natural numbers *Nat* and the object identifiers *Obj*, including *this*, which may be passed as arguments to methods.

Processor Release Points. Guarded commands g are used to explicitly declare potential processor release points **await** g. Guarded commands can be nested within the same local variable scope, corresponding to a series of processor release points. When an inner guard which evaluates to false is encountered during process execution, the process is suspended and the processor released. After processor release, any suspended process may be selected for execution. The type *Guard* is constructed inductively:

- $wait \in Guard$ (explicit release)
- $t? \in Guard$, where $t \in Label$
- $b \in Guard$, where b is a boolean expression over local and object state
- $g_1 \wedge g_2$ and $g_1 \vee g_2$, where $g_1, g_2 \in Guard$.

Use of *wait* will explicitly release the processor. The reply guard $t?$ succeeds if the reply to the method invocation with label t has arrived. Evaluation of guards is done atomically.

Internal control flow in objects is expressed by composing guarded commands. Let GS_1 and GS_2 be guarded commands **await** $g_1; \mathrm{S}_1$ and **await** $g_2; \mathrm{S}_2$. Inner guards are obtained by sequential composition; in the statement $GS_1; GS_2$, the guard g_2 is a potential release point. *Synchronized merge*, $GS_1 \& GS_2$, is defined as **await** $g_1 \wedge g_2; \mathrm{S}_1; \mathrm{S}_2$, treating non-guarded arguments as guarded by true and expanding synchronized method calls. Control flow without potential processor release uses standard **if** and **while** constructs, and assignment to local and object variables is expressed as $\mathrm{V} := \mathrm{E}$. In-parameters as well as *this*, *label*, and *caller* are read-only variables, the two latter local to method instances.

With nested release points, the object need not block while waiting for replies. This approach is more flexible than future variables: suspended processes or new method calls may be evaluated while waiting. If the called object never replies, deadlock is avoided as other activity in the object is possible. However, when the reply arrives, the *continuation* of the process must compete with other enabled suspended processes.

Example: Inheritance of synchronization constraints. The language constructs are now illustrated by examples from the literature on the inheritance anomaly [24], in particular anomalies related to the use of guards. First, we provide a general class *LRec* (Limited Resource) implementing a general controlling mechanism for resource allocation.

```
class LRec(limit: Nat)
begin var cnt: Nat = 0
  op incr(n: Nat) == await (cnt + n ≤ limit); cnt := cnt + n
  op decr(n: Nat) == await (cnt - n ≥ 0); cnt := cnt - n
end
```

Let unbounded buffers be defined by a class *Buf* with unguarded operations *put*(**in** *x*: *Data*) and *get*(**out** *x*: *Data*). We assume that a *get* operation returns some kind of error message if it is invoked on an empty buffer. By means of *multiple inheritance* and *synchronous merge*, these two classes may be used to implement a bounded buffer using waiting, in order to perform buffer operations in a safe manner.

```
class Buf1(lth: Nat) inherits LRec(lth), Buf
begin
  op put (in x: Data) == incr_LRec(1) & put_Buf (x)
  op get (out x: Data) == decr_LRec(1) & get_Buf (;x)
  op get2 (out x1, x2: Data) == decr_LRec(2) & (get_Buf (;x1); get_Buf (;x2))
end
```

Note the *get2* method where the guard ensures that two *get* calls can be performed. Further, the synchronization constraints have been defined in a separate class, which may be reused in other contexts.

In the inheritance anomaly related to *history sensitive behavior*, an operation *gget* is added to the buffer class which should behave like *get* expect that it must wait after a normal *get*. We define a mix-in class *Lock*, with general synchronization operations. By using multiple inheritance, the lock is added to the buffer class and the buffer operations are redefined adding synchronization by means of synchronous merge:

```
class Lock
begin var locked: Bool=false
  op unlock == locked := false
  op lock == locked := true
  op sync == await (¬locked)
end
```

```
class Buf2(lth: Nat) inherits Buf1(lth), Lock
begin
  op put (in x: Data) == unlock_Lock & put_Buf1(x)
  op get (out x: Data) == lock_Lock & get_Buf1(;x)
  op gget(out x: Data) == sync_Lock & get_Buf1(;x)
end
```

This way, we have obtained a history sensitive version of the buffer class by combining the two superclasses in a clean manner. The resulting *gget* method is guarded by $(\neg \text{locked} \wedge \text{cnt} \geq 1)$, ensuring that both guards are satisfied before the operation may start. This is in general crucial to avoid deadlock: for instance if the *sync* operation grabs the lock a *gget* would then block a succeeding *gget*

```
op sync == await ¬locked; locked := true
```

This example shows again how business code and synchronization code can be developed independently, and combined effectively and cleanly. In contrast to recent aspect-oriented approaches [25], including synchronization patterns and composition filters, we use the same basic language to express both kinds of code.

3 Formal Reasoning in the Presence of Inheritance

We now consider invariant reasoning in presence of inheritance. In addition to the importance of inheritance as a code reuse mechanism, it is desirable to obtain reuse at the level of reasoning [27]. When a subclass redefines a method from a superclass, the ability to invoke the super method enhances understandability and clearness of the code. When reasoning about subclasses, we want to reuse existing proofs of the superclass invariants.

Class invariants. In a non-terminating system it is difficult to specify and reason compositionally about behavior in terms of pre- and postconditions. Instead, pre- and postconditions to method declarations are used to establish a *class invariant*. In order to facilitate compositional reasoning about Creol programs, the class invariant will be used to establish a *relationship between the internal state and the observable behavior* of class instances. The internal state reflects the values of class attributes and the observable behavior is expressed by a set of potential communication histories.

For this purpose, the class attributes are extended with a mythical history variable $\mathcal{H}$, using the empty (ε) and right append ($\vdash$) constructors for sequences, and the code is extended with mythical statements to update the history variable for every *output message* generated by the program code [10]. Especially, the history is updated with a completion message at the end of every method execution.

We introduce Hoare triples [5, 10, 16] on the form $\{P\}S\{Q\}$ where P and Q are predicates over program states and S is a statement list in the programming language. The meaning of such triples is that if S starts execution in a state fulfilling P and the execution of S terminates, it will do so in a state fulfilling Q. Let $FV[P]$ and $FV[S]$ denote the set of variables occurring free in a predicate P and a statement list S, respectively. The set of variables which may have their values changed by execution of S is denoted $\mathcal{W}[S]$, i.e. the set of variables occurring on the left hand side of an assignment statement or as actual out-variables in a synchronous call or reply.

The class invariant of C, denoted I_C, is defined over class attributes w_C and the history sequence, but not over variables local to each method instance: $FV[I_C] \subseteq \{w_C, \mathcal{H}\}$. The invariant must be proved to hold at object creation by inspection of the initial values of the class attributes. Consequently we may assume that I_C holds when the object processor starts execution of an invoked method, if we can verify that I_C holds every time processor control is released. Due to Creol's processor release points, this can happen in two ways:

- method execution is completed, or
- guards in **await** sentences are not satisfied, so method execution is suspended.

If we can establish I_C before processor release points, it will be safe to assume I_C every time a process starts or *resumes* execution. For a general method body S in C we then have the following verification condition: $\{I_C\}\,S\,\{I_C\}$. In addition, if I_C is established before a processor release point, we may assume I_C and validity of the guard afterwards [12]:

$$\{I_C\}\,\textbf{await}\,\phi\,\{I_C \wedge \phi\}$$

Our semantics allows *local reasoning* about objects in a concurrent system. The local history $\mathcal{H}_o$ of a given object o relates to the global history $\mathcal{H}_G$ by $\mathcal{H}_o = \mathcal{H}_G/o$, where $\mathcal{H}_G/o$ denotes the restriction of $\mathcal{H}_G$ to messages sent to or from o. All invariants I_C in this paper may refer to both input and output messages involving C, but may only restrict output, i.e.

$$\{I_C\}\,\mathcal{H} := \mathcal{H} \vdash \langle \mathit{Input\ message} \rangle\,\{I_C\}.$$

During a suspension of a particular process, other processes are allowed to capture processor control, which may lead to changes of object variables, thus $\mathcal{W}[\textbf{await}\,\phi] = w_C$. Also, the validity of a predicate not mentioning any of the variables which might have their values changed by execution of S, is not affected by the execution, which is formalized in the axiom CONS of Hoare Logic:

$$\text{CONS:} \quad \{P\}\,S\,\{P\} \qquad \text{where } FV[P] \cap \mathcal{W}[S] = \emptyset$$

Especially, this means that actual in-parameters and variables declared local to a method are not changed during process suspension.

Reasoning about synchronized merge. In this section we will develop sound reasoning rules for the synchronized merge construction in Creol. This development is based on analysis done in [12] which provides a sound reasoning system for sequential statements, guards and asynchronous method calls in Creol based on Hoare Logic and weakest liberal preconditions [11].

Given two statement lists $\textbf{await}\,g_1; s_1$ and $\textbf{await}\,g_2; s_2$ and assume that we have the following separate proofs for these: $\{I_1\}\,\textbf{await}\,g_1; s_1\,\{Q_1\}$ and $\{I_2\}\,\textbf{await}\,g_2; s_2\,\{Q_2\}$. We will now develop a reasoning rule for the combined construction

$$\{I_1 \wedge I_2\}\,(\textbf{await}\,g_1; s_1) \& (\textbf{await}\,g_2; s_2)\,\{Q_1 \wedge Q_2\}$$

The predicates Q_1 and Q_2 may be identical to the invariants I_1 and I_2 respectively, but this is not required. By using the definition of & and reasoning about initial guards, the above statement is reformulated as

$$\{I_1 \wedge I_2 \wedge g_1 \wedge g_2\}\,s_1; s_2\,\{Q_1 \wedge Q_2\}$$

Assuming $FV[I_2, g_2] \cap \mathcal{W}[s_1] = \emptyset$ and $FV[Q_1] \cap \mathcal{W}[s_2] = \emptyset$, the axiom CONS gives

$$\{I_2 \wedge g_2\}\,s_1\,\{I_2 \wedge g_2\} \quad \text{and} \quad \{Q_1\}\,s_2\,\{Q_1\}$$

Using sequencing and left weakening we combine the last three conditions and get the following two premises:

$$\{I_1 \wedge g_1\}\, s_1\, \{Q_1\} \quad \text{and} \quad \{I_2 \wedge g_2\}\, s_2\, \{Q_2\}$$

which are provable by assumption. We may then conclude with the rule:

$$\frac{\{I_1\}\, \textbf{await}\, g_1; s_1\, \{Q_1\} \qquad \{I_2\}\, \textbf{await}\, g_2; s_2\, \{Q_2\}}{\{I_1 \wedge I_2\}\, (\textbf{await}\, g_1; s_1) \& (\textbf{await}\, g_2; s_2)\, \{Q_1 \wedge Q_2\}} \tag{1}$$

where $FV[Q_1] \cap \mathcal{W}[s_2] = \emptyset$ and $FV[I_2, g_2] \cap \mathcal{W}[s_1] = \emptyset$. If s_1 is empty, our first assumption reduces to $\{I_1\}\, \textbf{await}\, g_1\, \{I_1 \wedge g_1\}$. By a corresponding argument we then conclude with:

$$\frac{\{I_2\}\, \textbf{await}\, g_2; s_2\, \{Q_2\}}{\{I_1 \wedge I_2\}\, (\textbf{await}\, g_1) \& (\textbf{await}\, g_2; s_2)\, \{I_1 \wedge g_1 \wedge Q_2\}} \tag{2}$$

where $FV[I_1, g_1] \cap \mathcal{W}[S_2] = \emptyset$ In the next subsection, we will use these general rules to derive sound rules for inheritance reasoning.

Combining Invariant Reasoning and Inheritance

Using the notation m_C for synchronous invocation of a method m in a superclass C, redefinitions of super methods may be combined with the operator &, so the sentence $n_{C_1} \& \, m_{C_2}$ invokes n as declared in C_1 and m declared in C_2. Execution of this sentence leads to a suspension if at least one of the initial guards in n or m is not fulfilled.

Reasoning about class invariants and single inheritance. When a subclass invokes a method defined in a superclass C, the subclass is responsible for establishing I_C before the invocation, and may consequently assume I_C after the invocation. This means that for every method m defined in C, the subclass may assume:

$$\{I_C\}\, m_C\, \{I_C\}$$

After an invocation, the subclass is further responsible for establishing its own invariant before the next processor release point.

Reasoning about multiple inheritance. Mix-in classes are used to combine behavior and synchronization constraints, so we assume that two classes C_i and C_j inherited by a subclass C have no variables defined in a common ancestor class. Given k mix-in classes $C_1 ... C_k$ with invariants $I_1 ... I_k$ and assume that these classes are inherited by a common successor class C. The side conditions for the rules (1) and (2) are then fulfilled, or more precisely $FV[I_j, n_{C_j}] \cap \mathcal{W}[m_{C_i}] = \emptyset$ for $i \neq j, 1 \leq i \leq k$ and $1 \leq j \leq k$. As for single inheritance, C may assume $\{I_i\}\, m_{C_i}\, \{I_i\}$ for every method m in C_i and every i: $0 \leq i \leq k$. Let m be a method in C_i and n be a method in C_j for $i \neq j$. As a special case of (1) and (2) we may then reason about composition of super calls as follows:

$$\text{INH}_1 : \frac{\{I_i\}\, m_{C_i}\, \{I_i\} \qquad \{I_j\}\, n_{C_j}\, \{I_j\}}{\{I_i \wedge I_j\}\, m_{C_i} \,\&\, n_{C_j}\, \{I_i \wedge I_j\}} \qquad \text{INH}_2 : \frac{\{I_j\}\, n_{C_j}\, \{I_j\}}{\{I_i \wedge I_j\}\, (\textbf{await}\ \phi) \,\&\, n_{C_j}\, \{I_i \wedge I_j \wedge \phi\}}$$

where **await** ϕ is an expanded method body in C_i. It may be noted that although two different classes do not share write access to common variables, two invocations of super methods share *read* access to some variables. This includes *this* (identity of self) in addition to *caller* and *label*. The history sequence is also shared among the super classes, however local reasoning is preserved by restrictions to I_C.

4 Examples

We will in this section use the provided inheritance mechanisms to implement a version of the readers/writers problem and use the derived rules to reason about the implementation. The implementation uses two synchronization classes providing locking facilities for one and several clients, respectively. The singleton synchronization class will be inherited by the class *WSync*, giving writers exclusive right to the writing facility. The multiple lock will be inherited by the *RSync* class, giving an arbitrary number of readers access to the reading facility. At last, *WSync* and *RSync* are inherited by a class *RWSync* which provides synchronization between readers and writers to ensure mutual exclusion.

Interfaces. The interfaces *ReadOnly* and *ReadWrite* provides basic reading and writing activity and will not be further specified, but assumed to be implemented in a class *RW*. In addition, we specify the two interfaces *Acc* and *Sync* used to implement locking facilities.

```
interface Acc
begin
  op open()
  op close()
end
```

```
interface Sync
  inherits Acc
begin
  op sync()
  op waitFree()
  inv Ok(H)
end
```

```
interface ReadOnly
begin
  op read(in k: Key out y: Data)
end
```

```
interface ReadWrite
  inherits ReadOnly
begin
  op write(in k: Key, x: Data)
end
```

Using inheritance, the interface *Sync* provides four methods: *open*, *close*, *sync* and *waitFree* and respects the invariant specification $Ok(\mathcal{H})$. Let $m \in Mtd$, and let $caller \leftarrow m$ be short notation for messages sent from *this* to *caller* at completion of the method m. The interface invariant is then defined as a predicate over the history:

$$\begin{aligned}
Ok(\varepsilon) &== \mathsf{true}\\
Ok(\mathcal{H} \vdash caller \leftarrow waitFree) &== Ok(\mathcal{H}) \wedge Lock(\mathcal{H}) = \emptyset\\
Ok(\mathcal{H} \vdash caller \leftarrow sync) &== Ok(\mathcal{H}) \wedge caller \in Lock(\mathcal{H})\\
Ok(\mathcal{H} \vdash \textbf{others}) &== Ok(\mathcal{H})
\end{aligned}$$

An occurrence of **others** matches all events not giving a match against any of the above left hand sides. In this case **others** will match completions of *open* and *close*, as well as input to the interface which means that the interface lies no restriction on calls made by the environment. The function $Lock : Seq[Msg] \rightarrow Set[Obj]$ is defined over message sequences and returns the set of clients which captures the lock:

$$\begin{aligned}
Lock(\varepsilon) &== \emptyset\\
Lock(\mathcal{H} \vdash caller \leftarrow open) &== Lock(\mathcal{H}) \cup \{caller\}\\
Lock(\mathcal{H} \vdash caller \leftarrow close) &== Lock(\mathcal{H}) \setminus \{caller\}\\
Lock(\mathcal{H} \vdash \textbf{others}) &== Lock(\mathcal{H})
\end{aligned}$$

Synchronization classes. The *Sync* interface will be implemented in two classes, *SSync* and *MSync*, which provide locking mechanisms. *SSync* allows only one client to capture the lock at a time, and *MSync* allows arbitrary many clients to share the lock. The method *open* returns when the client is given access to the lock, and this access is returned by an invocation of *close*. The method *sync* returns only when the calling client has access to the database and *waitFree* returns only when the lock is free. Let Oid denote the type of object identifiers.

```
class SSync implements Sync
begin var lock: Oid = null
  op open == waitFree(); lock := caller
  op close == if lock = caller
               then lock := null fi
  op sync == await (lock = caller)
  op waitFree == await (lock = null)
end
```

```
class MSync implements Sync
begin var mlock: OidSet = ∅
  op open == mlock := mlock ∪ {caller}
  op close == mlock := mlock \ {caller}
  op sync == await (caller ∈ mlock)
  op waitFree == await (mlock = ∅)
end
```

The abstract description of *Sync* can be related to the internal state of objects of the *SSync* and *MSync* classes by the following invariants:

$$I_{SSync}: \ Lock(\mathcal{H}/SSync) = \{lock\} \qquad I_{MSync}: \ Lock(\mathcal{H}/MSync) = mlock$$

where $\mathcal{H}/MSync$ and $\mathcal{H}/SSync$ denote the restrictions of the global communication history to events involving only *MSync* and *SSync* respectively.

Readers. Readers may share access to the database, so we use the *MSync* class to synchronize them. All the methods defined in *MSync* are inherited directly, but *read* operations are only allowed to take place when the client is registered in the lock. This is achieved by invoking $sync_{MSync}$ before reading can take place.

A reading session related to a specific client then takes place by an invocation of *open* followed by an number of *read* operations, and is completed by an invocation of *close*. Note that this implementation allows a client to invoke the *read* operation even if the client is not registered in the lock, the invocations will then be suspended due to the synchronization requirement. Eventually, when the client has signed up for the lock, the suspended *read* operations can take place. Invocation of the *write* method defined in *RW* is prohibited by interface specifications. This class inherits I_{MSync} as well as any invariant provided by *ReadOnly*. Concentrating on synchronization aspects, the latter is left unspecified, so $I_{RSync} == I_{MSync}$.

Writers. Writers are synchronized by *SSync* in order to get exclusive right to the database. The invariant thereby becomes $I_{WSync} == I_{SSync}$. Since the lock in *SSync* is implemented as an object identifier, typing information establishes the requirement that only one client can have write access at a time.

```
class RSync inherits MSync, RW
  implements ReadOnly, Acc
begin
  op read(in k: Key out y: Data) ==
         $sync_{MSync}$ & $read_{RW}$ (k; y)
end
```

```
class WSync inherits SSync, RW
 implements ReadWrite, Acc
begin
  op write(in k: Key, x: Data) ==
         $sync_{SSync}$ & $write_{RW}$ (k, x)
  op read(in k: Key out y: Data) ==
         $sync_{SSync}$ & $read_{RW}$ (k; y)
end
```

Note that the *sync* and *waitFree* methods are not available to the environment since they are not visible through the given interfaces.

Synchronization of Readers and Writers. At last, we join reading and writing capabilities in a subclass *RWSync* which inherits both *RSync* and *WSync*. By making calls to super methods combined with the & operator, we ensure that a client is only given read access when there are no writers and vice versa. Reading and writing operations are then synchronized by the following class:

```
class RWSync inherits RSync, WSync
  implements RWI
begin
   op openRead == waitFree_WSync & open_RSync
   op closeRead == close_RSync
   op openWrite == waitFree_RSync & open_WSync
   op closeWrite == close_WSync
   op read(in k: Key out y: Data) == read_RSync(k; y)
   op write(in k: Key, x: Data) == write_WSync(k, x)
end
```

```
interface RWI
  inherits ReadWrite
begin
   op openRead()
   op closeRead()
   op openWrite()
   op closeWrite()
end
```

Mutual exclusion between readers and writers is guaranteed by the following invariant:

$$I_{RWSync}: \; I_{RSync} \wedge I_{WSync} \wedge (lock = null \vee mlock = \emptyset)$$

To verify *openRead*, we may assume the following from the superclasses:

$$\{I_{WSync}\}\, waitFree_{WSync} \{I_{WSync} \wedge lock = null\} \quad \text{and} \quad \{I_{RSync}\}\, open_{RSync} \{I_{RSync}\}$$

Combining these using INH$_2$ gives the following condition:

$$\{I_{WSync} \wedge I_{RSync}\}\, waitFree_{WSync} \,\&\, open_{RSync} \{I_{WSync} \wedge I_{RSync} \wedge lock = null\}$$

which proves the desired conclusion $\{I_{RWSync}\}\, openRead \{I_{RWSync}\}$ by consequence. Verification of *openWrite* can be done the same way. Note that the alternative definition of *openWrite* given by $waitFree_{RSync}; open_{WSync}$ leads to *undesired behavior* since the two guards involved must be satisfied *simultaneously* in order to enforce mutual exclusion.

Bounded sequential synchronization. We now combine the *SSync* class and the *Buf1* class given in Section 2 to form a class which performs synchronized *put* and *get* operations, such that there is a bound on the number of operations a client may perform before its permissions are withdrawn. The class *LimSSync* redefines the earlier given *SSync* in a way that restricts the holder of the lock to do a limited number of operations:

```
class LimSSync(limit: Nat) inherits SSync implements Sync
begin var count: Nat = 0
   op open == open_SSync &  count := 0
   op sync == sync_SSync &  (count := count + 1; if count = limit then close_SSync fi)
end
```

The invariant for this class strengthens the previous invariant I_{SSync}:

$$I_{LimSSync}: \;\; I_{SSync} \wedge 0 \leq count \leq limit$$

An invariant for *LRec* (and therefore also for *Buf1*) can be formulated as:

$$I_{Buf1}: \;\; 0 \leq cnt \leq lth$$

These two classes may again be inherited by a class *BufSync* providing locking mechanisms on the buffer, such that each client can only do a limited number of operations on the buffer before the lock is released.

```
class BufSync(lth: Nat, limit: Nat) inherits LimSSync(limit), Buf1(lth)
begin
  op put(in x: Data) == sync_LimSSync & put_Buf1(x)
  op get(out x: Data) == sync_LimSSync & get_Buf1(;x)
end
```

The invariant for this class becomes $I_{BufSync} : I_{Buf1} \wedge I_{LimSSync}$ which is proved by combining proofs of I_{Buf1} and $I_{LimSSync}$ using the rule INH$_1$.

Note that execution of *BufSync* may lead to deadlock, for instance if the client capturing the lock only invokes *put*. This limitation can be overcome by a more sophisticated synchronization class allowing a number of active clients, but keeping the ability to exclude clients from the buffer if they break some criteria of well behavior. This requires more detailed programming of *LimSSync*, not affecting the proof outline.

5 Related Work

Many object-oriented languages offer constructs for concurrency. A common approach has been to rely on the tight synchronization of RPC and keep activity (threads) and objects distinct, as done in Hybrid [26] and Java [15], or on the rendezvous concept in concurrent objects languages such as Ada [4] and POOL-T [3]. For distributed systems, with potential delays and even loss of communication, these approaches seem less desirable. Hybrid offers *delegation* as an explicit programming construct to (temporarily) branch an activity thread. Asynchronous method calls can be implemented in e.g. Java by explicitly creating new threads to handle calls [9]. To facilitate the programmer's task and reduce the risk of errors, implicit control structures based on asynchronous method calls seem more attractive, allowing a higher level of abstraction in the language.

Languages based on the Actor model [1, 2] take asynchronous messages as the communication primitive for loosely coupled processes. This makes Actor languages conceptually attractive for distributed programming. Representing method calls by asynchronous messages has lead to the notion of future variables found in e.g. languages such as ABCL [28], Eiffel// [8], CJava [9], and in the Join-calculus [14] based languages Polyphonic C$^\sharp$ [6], and JoinJava [19]. The Creol notion of asynchronous method calls and nested processor release points further extend this approach to asynchrony.

Most languages supporting asynchronous methods either disallow inheritance [19, 28] or impose redefinition of asynchronous methods [8]. In Polyphonic C$^\sharp$ inheritance is expressed as a disjunction of join patterns [14], resulting in nondeterminism rather than overloading, and supplemented by a substitution mechanism for inherited code. CJava [9], restricted to outer guards and single inheritance, allows synchronization code and bodies to be redefined separately in subclasses. The approach of Ferenczi [13] is similar to ours by using guards to reduce the inheritance anomaly, but only outer guards are considered, and there is no explicit merge operator like our &. However, there is a form of implicit merge by interpreting the sequential composition of two guards as a single guard with two conjuncts. The approach of Ferenczi may be used to add guards to inherited code, but not to define synchronization code in separate classes or modules.

Recent aspect-oriented approaches use constructions such as composition filters [7], synchronization rings [17], synchronization patterns [23], and even temporal logic [25] to achieve modular separation of business and synchronization code. An appealing property of our approach is that the same language constructs are used for both kinds of code.

6 Conclusions

The Creol language uses guards as a high level construct for synchronization and for controlling asynchronous interaction. In particular, a caller may decide whether to invoke a method asynchronously or asynchronously, whereas method declarations resembles those of synchronous methods. In this setting the combination of multiple inheritance, synchronous merge, and synchronous calls to superclass method is exploited to accommodate mix-in classes, separating synchronization code from business code while using the same language constructs. We have demonstrated through examples how the use of synchronous merge combined with synchronous local calls may be used to reduce the inheritance anomaly. The use of synchronous merge, which is introduced in this paper, gives simple solutions to the standard buffer anomaly examples.

We have also discussed reasoning issues related to our approach, and have shown that the synchronous merge is semantically clean when used as suggested. In particular we have a guarantee that superclass invariants are not violated when synchronous merge is used as a combinator for superclass methods from disjoint superclasses. This means that our approach facilitates reuse of partial correctness reasoning in subclasses. The interface level of Creol, not emphasized here, supports aspect-oriented and behavioral specification as well as compositional reasoning [20].

References

[1] G. A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, eds, *1st IFIP Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'96)*, p. 135–153. Chapman & Hall, 1996.

[2] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, Jan. 1997.

[3] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, eds, *Object-Oriented Concurrent Programming*, pages 199–220. The MIT Press, 1987.

[4] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

[5] K. R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.

[6] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\sharp$. In B. Magnusson, editor, *Proc. 16th Eur. Conf. on Object-Oriented Programming (ECOOP 2002)*, LNCS 2374, pages 415–440. Springer, June 2002.

[7] L. Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994.

[8] D. Caromel and Y. Roudier. Reactive programming in Eiffel//. In J.-P. Briot, J. M. Geib, and A. Yonezawa, editors, *Proc. Conf. on Object-Based Parallel and Distributed Computation*, LNCS 1107, pages 125–147. Springer, 1996.

[9] G. Cugola and C. Ghezzi. CJava: Introducing concurrent objects in Java. In M. E. Orlowska and R. Zicari, editors, *4th Intl. Conf. on Object Oriented Information Systems (OOIS'97)*, pages 504–514. Springer, 1997.

[10] O.-J. Dahl. *Verifiable Programming*. Prentice Hall, 1992.

[11] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[12] J. Dovland, E. B. Johnsen, and O. Owe. A Hoare logic for objects with asynchronous method calls. Research report, Dept. of Informatics, Univ. of Oslo, July 2004. In preparation.

[13] S. Ferenczi. Guarded methods vs. inheritance anomaly: Inheritance anomaly solved by nested guarded method calls. *ACM SIGPLAN Notices*, 30(2):49–58, Feb. 1995.

[14] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join calculus. In S. Kapoor and S. Prasad, eds, *20th Conf. on Foundations of Software Technology and Theoretical Comp. Science (FST TCS 2000)*, LNCS 1974, pages 397–408. Springer, 2000.

[15] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java language specification*. Java series. Addison-Wesley, Reading, Mass., 2nd edition, 2000.

[16] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[17] D. Holmes. *Synchronization Rings – Composable Synchronization for Object-Oriented Systems*. PhD thesis, Macquarie University, 1999.

[18] International Telecommunication Union. Open Distributed Processing - Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.

[19] G. S. Itzstein and M. Jasiunas. On implementing high level concurrency in Java. In A. Omondi and S. Sedukhin, eds, *19th Annual Computer Security Applications Conference (ACSAC 2003)*, LNCS 2823, pages 151–165. Springer, 2003.

[20] E. B. Johnsen and O. Owe. A compositional formalism for object viewpoints. In B. Jacobs and A. Rensink, editors, *Proc. 5th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*, pages 45–60. Kluwer Academic Publishers, Mar. 2002.

[21] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd IEEE Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE Computer Society Press, Sept. 2004.

[22] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous methods calls. In *Proc. 5th Intl. Workshop on Rewriting Logic and its Applications (WRLA'04)*, To appear in ENTCS, Elsevier, 2004.

[23] C. V. Lopes and K. J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In M. Tokoro and R. Pareschi, editors, *Object-Oriented Programming (ECOOP'94)*, LNCS 821, pages 81–99. Springer, 1994.

[24] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, eds, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[25] G. Milicia and V. Sassone. The inheritance anomaly: ten years after. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1267–1274. ACM Press, 2004.

[26] O. Nierstrasz. A tour of Hybrid – A language for programming with active objects. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 167–182. Prentice Hall, 1992.

[27] N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. 5th Intl. Conf. on Software Reuse (ICSR5)*, pages 206–215. IEEE Computer Society Press, 1998.

[28] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. The MIT Press, 1990.