Integrating Concurrency and Object-Oriented Programming: An Evaluation of Hybrid

KONSTANTAS, Dimitri, PAPATHOMAS, Michael

Abstract

In this paper we address the effective use of the object-oriented programming approach for concurrent programming from a language design viewpoint. We present a set of requirements for the design of concurrent object-oriented languages. We then use a particular language, Hybrid, as a concrete example and examine to what extent its features meet these requirements. We identify the solutions offered by Hybrid and its shortcomings and we underline both the difficulties and promising directions for the design of concurrent object-oriented languages.

Reference


Available at:
http://archive-ouverte.unige.ch/unige:72834

Disclaimer: layout of this document may differ from the published version.
The requirements that we formulated for concurrent object-oriented languages provide some guidelines for the design of languages and their mechanisms. However, a framework providing a more rigorous formulation of the requirements and the evaluation of mechanisms would be extremely helpful for the development of better language designs. Our current research [11] is directed towards the development of such a framework as well as an improved version of Hybrid.

References


are not interchangeable (unless the object and its clients are in different domains) because of the use of delegated calls. It would not be possible to upgrade a sequential implementation of an object to a parallel one without affecting the clients of the object (unless it can be determined that the object to be upgraded is never in the same domain as its callers).

7.3 Scheduling Requests

There are several cases where objects need to service requests in an order that is different from the order in which they arrive. The resource allocator, in section 5., showed that one reason for doing this is that it may not be possible to service a request at a certain point in time. For instance, when not enough resources are available, allocation requests may only be processed after deallocation requests despite their order of arrival. Another reason is that it may be more efficient to serve requests in a certain order. A disk head scheduler is a typical example. The possibility to service requests in an order that minimizes the disk head’s movements has substantial efficiency gains.

The general solution for scheduling requests in Hybrid, presented in section 5.3, uses a combination of delay queues and delegated calls. Because of this, scheduling of requests is an implementation decision that may not, in general (only if the object is not in the same domain as its clients), be made transparently to an object’s clients.

8. Conclusion

Hybrid served as a concrete example showing how some aspects in the design of a concurrent object-oriented language satisfy or fail to satisfy the requirements stated in section 3. The examination of Hybrid’s features allows us to draw general conclusions about the design of concurrent object-oriented languages and the integration of concurrency and object-oriented programming.

The comparison of Hybrid’s active view of objects to other approaches showed that this approach is the most suitable for object-oriented programming. It is the most appropriate for supporting encapsulation and data abstraction. Moreover, considering all objects as active server-like entities presents higher re-use potential than distinguishing among data and active objects and may be implemented at least as efficiently.

The concurrency mechanisms of Hybrid are well suited for concurrent programming and more particularly for the client-server structure of object-oriented programs. However, in section 7, we presented examples that revealed some limitations in their support for data abstraction. In particular, in order to correctly reuse objects it has to be known whether delegated calls are used in their implementation. Moreover, objects that are implemented by using delegated calls are not, in general, substitutable with those that do not, despite the fact that they may present the same interface and provide the same services to their clients.

An important observation is that the limitations of Hybrid’s concurrency mechanisms for supporting data abstraction was demonstrated by bringing up the appropriate examples. Obtaining such examples for evaluating the design of a language is not always easy and sometimes only possible after the language has been implemented and used.
exact way in which it is implemented. Furthermore, conforming to abstract typing, it should be possible to interchange objects with compatible interfaces and implementations.

The delegated call mechanism of Hybrid fails to meet this requirement. It is not possible to substitute an object with one that uses delegated calls in its implementation, despite the fact that it has the same interface and provides the same services. Below, we first explain what in the semantics of the delegated call mechanism causes this problem. Then, we discuss two practical cases where, because of delegated calls, it is not possible to hide certain implementation choices from objects’ clients. These are the use of delegated calls for executing requests in parallel and for scheduling requests.

7.1 The Delegated Call Problem

When an object issues a delegated call its domain becomes idle a thread may then be triggered for processing a request addressed to any object in the domain. This has the effect that the execution of a delegated call in an object may trigger the execution of a thread in another object in the domain.

Consider an object $A$ which calls a method of another object $B$ which is in the same domain. If $B$ issues a delegated call, it may trigger the execution of a method of another object in its domain. In particular, the delegated call may trigger the execution of another method of $A$. This causes an interleaved execution of $A$’s methods since the method that invoked $B$ in the first place is not yet completed. This interleaved execution of $A$’s methods, which occurs in points that are not controlled by $A$ but by objects that are called by it, may cause the execution of $A$’s method to fail. There is no guarantee that a class invariant, which is a necessary precondition for the correct execution of $A$’s methods, holds when $A$ calls $B$ or any other object in the same domain that performs a delegated call.

This problem may be circumvented by an appropriate implementation of $A$ that takes into account the fact that delegated calls are used in $B$. It is possible to implement $A$ in such a way that the class invariant is satisfied before calling $B$’s methods or to prevent the activation of other $A$’s methods by using delay queues or by having $A$ and $B$ be in separate domains. These solutions may be inconvenient and rely on the fact that it is known that $B$’s implementation uses delegated calls. This is contrary to data abstraction, since it is necessary to know how an object is implemented in order to construct correct programs. Furthermore, it is not possible to safely replace the implementation of an object that didn’t use delegated calls by one that does without re-examining all the programs that were based on the old implementation.

7.2 Parallel Execution of Requests

As only one thread at a time may be active in a Hybrid domain, the only way to implement objects that process requests in parallel is by using several objects, each in a separate domain, that cooperate in order to realize the required functionality.

In Hybrid, the administrator example, discussed in section 5.2, is a typical way of implementing an object that executes requests in parallel. However, such an implementation makes use of delegated calls. This means that two implementations of an object, the one sequential and the other parallel, with the object implemented as an administrator managing a set of workers,
6. Using Domains and Activities For Coordinating Objects

Computations often involve collections of cooperating objects. The correctness of the computation is based on assertions that are made by an object about the state of another object. The concurrent execution of objects may cause state changes that invalidate such assertions.

Consider the following example. A resource allocator object manages a number of reusable resources. It has the methods: `request`, `release` and `atHand` which are used for requesting, releasing a number of resources and inquiring about the number of resources that are available. Requests made by objects for more than the currently available resources are delayed until enough resources are released. Assume that for the needs of some application, an object should not get suspended if its request may not be handled immediately. A way to handle this without modifying the allocator would be for the object to first find out whether enough resources are available by invoking the `atHand` method, and then invoke the `request` method. However, requests issued by concurrently executing objects may get processed in between. So because of concurrent execution the assertion made by the client that enough resources are available may be invalid by the time that `request` is executed by the allocator.

The domain and activity concepts provide the possibility to coordinate the execution of objects so that no interference may occur by their concurrent execution. Moreover, it is not required for the objects to be coded in a special way or to modify them for achieving the desired synchronization. In the above example, it is enough for the allocator object and the client that shouldn’t wait to be in the same domain. In this case, the allocator may not accept requests from other activities while an activity is active within the domain. Thus, no other calls to `request` may be accepted by the allocator between the calls to `atHand` and `request` made by the client.

The domain and activity concepts provide the possibility to coordinate the execution of objects so that no interference may occur by their concurrent execution. Moreover, it is not required for the objects to be coded in a special way or to modify them for achieving the desired synchronization. In the above example, it is enough for the allocator object and the client that shouldn’t wait to be in the same domain. In this case, the allocator may not accept requests from other activities while an activity is active within the domain. Thus, no other calls to `request` may be accepted by the allocator between the calls to `atHand` and `request` made by the client.

The concepts of domain and activity are very useful when it is needed to coordinate the execution of a collection of objects. However, domains and activities present some limitations. The domains of objects are specified at the time they are created and cannot be changed during their lifetimes. In the allocator example presented above, once the allocator and the client are created in the same domain they will never be able to execute concurrently. This is overconstraining since there are cases where they could perfectly do so.

We are investigating extensions of the domain and activity concepts that, while retaining their advantages, would be more flexible and permit more parallelism. For instance, a useful extension would be to allow objects to join and leave domains dynamically. In the allocator example, the client would request the allocator to join its domain just for the execution of the method that calls `atHand` and `request`. After that, the allocator and client would be able to execute concurrently.

7. Support For Data Abstraction

In our requirements for the concurrency constructs of object-oriented languages presented in section 3. we stated that the concurrency constructs should be compatible with and provide support for data abstraction. In particular, it should be possible for clients of an object to ignore the
The main disadvantages of this approach are that the code of the administrator becomes rather complex and that the objects acting as workers have to be designed especially so that they may cooperate with the administrator. Concerning the latter, the replies to requests have to be returned by asynchronous messages that should contain some information allowing the administrator to associate a reply to a request. This may be avoided by mechanisms such as ConcurrentSmalltalk’s CBoxes[15] or ABCL/1’s future type message passing[16]. With these mechanisms the objects that may be used as workers by the administrator do not have to be designed in any special way.

Early reply is a technique that is used for simulating asynchronous message passing when it is not supported by the language. The main idea is that the administrator returns immediately to the client the identifier of another object which it should call for getting the result. This allows the administrator to accept requests from other clients before it has processed the first client’s request. The same technique can be used for the interaction of the administrator and workers so that the administrator is not blocked during the time that a worker processes a request.

The main problem with this technique is that it requires a specific interaction protocol to be observed between objects. A client object must be prepared to cope with early reply by first obtaining the identity of the object to call for getting the result of its request and then calling this object to effectively get hold of the result. Concerning clients, an object implemented following the administrator paradigm presents a different more complex interface than objects with a sequential implementation. This is contrary to data abstraction and has the disadvantage that objects implemented following the administrator paradigm cannot be used in the place of other objects.

Multi-Thread Objects

The dynamic creation of threads within objects for the execution of requests eliminates the problems of explicitly multiplexing the single object thread among several requests. However, the concurrent execution of threads within the object presents synchronization problems due to their access to shared instance variables. If these threads are created dynamically whenever a method is invoked, it will be necessary to explicitly synchronize the execution of methods. Such an approach is similar to the orthogonal approach, discussed in section 4., and provides limited support for encapsulation, data abstraction and object reuse.

The delegated call mechanism presented in section 5.2 provides a satisfactory compromise between the above alternatives. It avoids the problems due to the explicit multiplexing of the single object thread by using a separate thread for each request. Moreover, by providing more control on the creation and concurrent execution of threads within objects it avoids the problems of the orthogonal view of objects. However, by requiring that only one thread be active within an object, this approach unnecessarily restricts concurrency in some cases. For instance, multiple requests that only read an object’s state cannot be processed in parallel.
there is a delayed call. If not, the delay queue would remain open and the next call to suspend would not be delayed.

5.4 Comparison to Other Approaches

5.4.1 Local Delays

Delay queues combined with delegated calls, in the way presented in section 5.3, can handle any local delay. However, it would be more convenient if we were able to do it in a more compact and elegant way. Moreover, as we will show in section 7, this way of handling local delays does not combine well with data abstraction.

Other alternatives for local delays are condition variables and selective acceptance of requests with guards, that may depend on method arguments, as in the programming language SR[3]. Condition variables are used in monitors[6] to suspend the execution of a monitor procedure. This may take place after testing a condition that may depend on both the object’s state and method arguments. Guards are boolean expressions that are associated with a method. Requests for invoking a method are accepted only when the associated guard is true.

Delay queues resemble a restricted form of guard that tests whether the associated delay queue is open. The combination of delay queues and delegated calls used in section 5.3 resembles the use of condition variables in monitors.

The easiest way in which delay queues can be extended in the current design and implementation of Hybrid is by adding an operation “wait” to the delay queue type. This operation would suspend the invoking thread if a condition necessary for its execution is not satisfied. This is similar to the use of condition variables in monitors.

5.4.2 Remote Delays

The issues concerning remote delays differ depending on whether or not the language supports single threaded objects. In the case where objects are single threaded or there is a fixed number of threads per object, the main issue is how to multiplex an object’s thread among several requests. In the case where a thread may be created for each request the problem is the synchronization of threads within the object. We further discuss these issues below based on the administrator example from section 5.2.

Single Thread Objects

With single thread objects remote delays may be handled by using asynchronous message passing or, in the case where asynchronous message passing is not supported, by a technique called early reply[9].

With asynchronous message passing the administrator object issues requests to workers by asynchronous messages. This allows it to accept and process more client requests. The replies to the requests made to the workers are returned to the administrator by another asynchronous message. These messages identify the requests so that the administrator may associate the replies with the corresponding requests. The administrator should be programmed in such a way that it is able to accept new client requests as well as replies returned by workers.
pass on the requested work by issuing a delegated call

```csharp
result <- delegate(@workers[workerIndex].doIt(work));
gather some statistics about workers based on result
return the result to the caller.
```

```csharp
return(result);
```
...insert an item
  if the buffer is now full
    putDelay.close();
    getDelay.open();
  }
}

The delay queues `getDelay` and `putDelay` are used for avoiding the acceptance of get and put requests when the buffer is empty and full respectively. Initially the delay queue `getDelay` is closed so that no get requests are accepted and the delay `putDelay` is open so that put requests are accepted and items may be inserted in the buffer. When the method `put` is executed it opens the delay queue `getDelay` so that get requests may be accepted. The method `get` determines whether the buffer is empty and then closes the `getDelay` delay queue. The method `put` closes the `putDelay` delay queue when the buffer is full.

In the above example, whether a request will lead to a local delay is determined by the requested method and the local state of the object. However, there are cases where this information is not sufficient and it is also necessary to know the values of the method’s arguments. Such cases are dealt with in Hybrid by combining delay queues with the delegated call mechanism. In section 5.3, after we have introduced the delegated call mechanism, we will show how to handle more complex cases by a combination of delay queues and delegated calls.

### 5.2 Remote Delays With Delegated Calls

The delegated call mechanism is well suited for remote delays. When an object issues a delegated call the executing thread is suspended. Another request may then be processed by another thread. A thread that was suspended because of a delegated call is resumed after the delegated call returns and there are no other active threads within the object.

As an example illustrating the use of the delegated call mechanism, consider an object that operates as an administrator for a collection of worker objects. This object is acquainted with a collection of worker objects that it uses for servicing the requests made by clients. Remote delays occur when the administrator invokes a particular worker for doing the job. It would not make sense for the administrator to wait until the worker finishes the requested work. Instead, it should accept other requests and pass them on to other workers that would process them in parallel. The use of the delegated call mechanism allows the administrator to switch its attention to another request while a call is passed on to a worker. The outline of the administrator is given below.

```plaintext
type administrator: abstract {
  doWork(work: WorkType)-> resultType;
} ;
private{
  var workers: array[1..MAX] of oid of workerType;
  doWork(work: WorkType) -> resultType
  {
    var workerIndex: integer;
    var result : resultType;

    select the worker appropriate for this work,
    assign its index to workerIndex.
```
structured as a collection of active modules acting as servers and clients. As this structure closely matches object-oriented programs we use these requirements for evaluating Hybrid’s concurrency features.

The main requirement formulated in [9] is that when an activity becomes blocked within a module (object in the case of Hybrid) other activities should be able to proceed. They identify two common situations where an activity may get blocked: *local delay* and *remote delay*.

Local delay is the situation where an object has to put aside a request because of the temporary unavailability of a resource. In such a case, the object should be able to set aside this request and turn its attention to others. It would be wasteful for the object to be blocked on a request while there are other requests that could be processed. Not being able to set aside a request that cannot be processed immediately may also cause unnecessary deadlocks. For instance, while being blocked because of the unavailability of a local resource, the object would be unable to accept requests that would make the resource available.

Remote delay is the situation in which an activity is delayed within an object because of a call to another “lower level” server. The delay may be due to the communication delay, the time needed by the lower level server for handling the request or a local delay in the lower level server. During this time further requests could be accepted and processed by the object.

### 5.1 Local Delays With Delay Queues

In Hybrid, simple forms of local delay are handled easily with delay queues. The use of delay queues allows objects to avoid accepting a request that would lead to a local delay. An illustration of the use of delay queues for simple cases of local delays is given below by the code outline of a bounded buffer.

```plaintext
type boundedBuffer : abstract {
    init : ->;
    put : string ->; uses delay;
    get : -> string; uses delay;
};
private {
    var putDelay, getDelay: delay;
    var buffer: strarray;
    ...
    init: ->;
    {
        ...putDelay.open();
        getDelay.close();
    }
    get: -> string; uses getDelay;
    {
        ...get an item.
        if the buffer is now empty
            getDelay.close();
            putDelay.open();
    }
    put: string ->; uses putDelay;
    {
```
object determines whether its methods need to be synchronized. Based on this information, the compiler generates synchronization code only in methods of “concurrent” objects. Thus, no synchronization overhead takes place for executing the methods of “data” objects. In an homogeneous approach all objects are of the same kind, so it may seem that synchronization code has to be generated in the methods of all objects.

In Hybrid the synchronization takes place when objects are invoked, rather than by code generated within methods. The use of different kinds of references for objects makes it possible to synchronize only the invocations of objects whose methods may be invoked concurrently.

The predefined object type, oid, is used for specifying a reference to an object. Oids may be passed around to other objects so that the methods of an object having an oid may be invoked concurrently. On the other hand, objects that are addressed by instance variable names are local to the enclosing object and are not known to other objects so their methods may not be invoked concurrently. As objects that are addressed by instance variable names may not be addressed by oids and inversely, only invocations through oids need to be synchronized.

The use of the type oid is illustrated below in the definition of a type A with a method anAMethod. Instances of this type have two instance variables, local and shared, for referencing two objects of a type B. The object stored in local may only be addressed through this name and thus it may not be known outside the scope of the containing object of type A. Variable shared stores an oid that is used to refer to another object of type B. The oid may be known to other objects, consequently invocations of objects addressed by oids, as @shared.aBMethod, are synchronized. No synchronization occurs when invoking methods of objects that are addressed by local names, as in local.aBMethod()?

```plaintext
5. Threads, Communication and Synchronization Primitives

Liskov et al. [9] have formulated concurrency requirements for evaluating the suitability of a combination of concurrency features for developing distributed programs. These programs are
```
4.1 Comparing the Approaches

The protection provided by encapsulation in the orthogonal approach does not protect objects from concurrent execution. The consistency of the internal state of objects may be compromised by the concurrent invocation of their methods. With the other two alternatives the internal state of objects is automatically protected from concurrent execution by extending encapsulation to cope with concurrency.

Compared to the orthogonal approach, the other two approaches better support data abstraction. With the orthogonal approach, whether an object will execute correctly in a concurrent application is determined by its implementation. The fact that the correct execution of objects depends on the way that they are implemented is contrary to data abstraction. With the alternative approaches, objects can be used in an application without concern for whether their implementation supports concurrent execution.

Consequently, the heterogeneous and homogeneous approaches have higher potential for reusability. On the one hand, no special effort is required in the implementation of a class whose instances may be used in a concurrent application. With the orthogonal approach it is necessary to explicitly include synchronization code in the methods. On the other hand, less effort is required for reusing a preexisting class. In contrast with the orthogonal approach, it is not necessary to determine whether the class implementation supports the concurrent invocation of methods of its instances.

The homogeneous approach has the advantages that the programmer does not have to decide whether an object should be implemented as a passive data type or as a “concurrent” type, and that parameterized code can be used with parameters bound to object types that have the appropriate interfaces. With the heterogeneous approach parameters cannot be bound equally to objects of different kinds even if they have the same interface.

The main motivation for a heterogeneous approach is to avoid, for “data” objects, the runtime overhead caused by the synchronization of methods of “concurrent” objects. With this approach, the object’s kind determines whether or not an object may be used in a concurrent context. This information is used by the compiler for generating the appropriate synchronization code. In the homogeneous approach, no such information is available to the compiler by object types so it may appear that all objects have to be implemented in a way that they may support concurrent execution. If this was the case it certainly would incur tremendous runtime overhead. Fortunately, there are other ways for the compiler to gather such information. Below we discuss how the design and implementation of Hybrid supports the homogenous approach with no more runtime overhead than the heterogeneous approach.

4.2 Supporting the Homogeneous Approach in Hybrid

Although in Hybrid all objects are considered as active entities that resemble server processes, they are implemented as passive data types. The active view of objects is supported by the appropriate synchronization of the objects’ methods.

The runtime overhead induced by the homogenous approach in this implementation is due to the synchronization of method execution. With the heterogeneous approach the “kind” of the
• **Data abstraction**: The concurrency constructs should make it possible for the clients to ignore the details of object implementations and for the implementors to be able to provide new compatible implementations without affecting the clients. This is necessary for supporting data abstraction and taking advantage of abstract typing for extensibility and code reuse.

• **Inheritance**: The design of the concurrency constructs of object-oriented languages should take inheritance into account for providing constructs that combine well with class inheritance. The integration of inheritance and concurrency in object-oriented languages is a difficult task. Kafura and Lee in [7] survey several languages (including Hybrid) and show how the concurrency constructs interfere with inheritance.

• **Promote object-oriented software development**: The concurrency constructs should be compatible with the client-server structure of object-oriented programs and facilitate the development of applications by reusing and adapting existing objects.

The discussion of the concurrency features of Hybrid that follows serves as a practical example by showing how it succeeds or fails to meet these requirements.

4. **The Object Model**

There are three main approaches to the design of concurrent object-oriented languages with respect to the way that the object model takes into account concurrency:

• The *orthogonal* approach, which considers concurrency independently of objects. Objects are the same as in sequential languages. The concurrency features may be used at will by the programmers in the implementation of objects’ methods. Some languages that have taken this approach are Smalltalk-80[9], Trellis/Owl[10] and Emerald[5].

• The *homogeneous* approach, which extends the notion of objects taking concurrency into account. This approach is exemplified by POOL-T[1] where objects are considered as single-threaded processes, and Hybrid where all objects are considered as multithreaded server-like entities.

• The *heterogeneous* approach which supports both data objects similar to sequential languages and “concurrent” objects that take into account concurrent execution. For example, ConcurrentSmalltalk[15] supports both ordinary Smalltalk objects and atomic objects which serialize the concurrent invocation of their methods. Avance[4] supports *packettypes* and *datatypes*. The execution of packettype methods by concurrent threads is serialized while datatypes are ordinary sequential objects.

Hybrid has followed the homogeneous approach by supporting an object model where all objects are considered potentially active server-like entities. This choice was guided by the fact that this approach, as we show below by comparing it to the other two alternatives, is better suited for object-oriented programming. The comparison of the approaches is based on their support for encapsulation, data abstraction and their potential for reusability.
In the above example every flight is represented as an independent object, that is a domain. The flightsControl object instantiates new flights and then initializes them, passing them the flight data (for example airplane type, date of flight etc.). The initialization is done via a reflex, so that a new thread of execution is started within the flight. Execution in the flight object is suspended by delegating an alarm call to the time object. This is done because we want to change the state of the flight just before departure. This way the flight object is free to serve other requests (for example information, reservations etc.).

3. Requirements For Concurrent Object-Oriented Languages

A great variety of programming notations [2] have been used in concurrent programming languages. The main requirements on these notations were to express parallel execution and the synchronization and communication of concurrent threads. Most notations are roughly equivalent, providing adequate expressive power for common concurrency problems. However, not all of them are equally well-suited for the same kind of computations. Operation oriented languages, which are based on remote procedure call communication primitives, are best suited for computations where processes interact as clients and servers. Message oriented languages, supporting one-way message passing, are best suited for pipelined computations. Also, distributed and concurrent programs that are structured as a collection of active modules acting as servers and clients impose specific requirements on the combination of the concurrency features of a language[9].

Although various approaches have been followed for the design of concurrent object-oriented languages not all of them are suitable for object-oriented programming[13]. Object-oriented programming imposes additional requirements on the concurrency features of programming languages. The main requirement is that the concurrency features of a language be compatible with object-oriented features such as encapsulation, data-abstraction and inheritance, and promote the development of applications by combining and tailoring already existing objects. The following points refine this main requirement.

- **Encapsulation:** Concurrent object-oriented languages should be designed in such a way that encapsulation ensures the integrity of the internal state of objects even when their methods are invoked concurrently.
sent to an object of another domain. Messages can be received by a domain only when it is idle. Messages sent to a domain that is in running or blocked state are queued and wait to be delivered when the object becomes idle. Messages waiting in the same queue are delivered on a first-come-first-served basis, while opened queues are scanned cyclicly.

A special type of operation is a reflex. Reflexes are operations that initiate a new activity by sending a start message to the remote object. Unlike call messages, no reply is expected from a start message. The sender will continue execution immediately after the invocation of the operation without waiting for the acceptance of the start message by the remote object. Reflexes can also be used in intra-domain communications. In this case the reflex is postponed until the calling activity is completed or sends a message that would leave the domain in an idle state (i.e. a delegated message, described bellow).

For the better support of active objects, Hybrid provides delegated calls for switching threads within domains in inter-domain communications. Any message sent via the delegation mechanism, will leave the calling domain in the idle state, allowing it to accept new messages. When the reply of the delegated message is received, execution will resume, as with any other message.

In order to provide a way for scheduling activities, Hybrid offers the mechanism of delay queues. Delay queues can be in two states controlled by the object, “open” and “closed” and are associated to operations. When a delay queue is closed, messages sent to it cannot be received, but rather are queued waiting for the delay queue to open. The domain is left idle and able to receive other messages. When the delay queue is opened, one of the queued messages is accepted.

The following example, an extract from a flight reservation control program, illustrates some of the features of Hybrid described above and gives an idea of what Hybrid looks like.

```plaintext
type Flight: abstract {
    reflex enableFlight ( ... ) -> ;
    reserve ( Passenger ) -> string ;
    cancel ( Passenger ) -> boolean ;
    ...
}

private {
    reserve (name : Passenger ) -> string ;
    {  ... }
    cancel (name : Passenger ) -> boolean ;
    {  ... }
    reflex enableFlight ( flightData : FlightData ) -> ;
    { ...
        delegate (@time.alarm( flightData.date ) ) :
            ...disable flight reservations 30 minutes before departure ...
    }
}

type FlightsControl : abstract {
    newFlight ( FlightData ) -> boolean ;
    reserve (Flight, Name ) -> string ;
    ...
```
In sections 4 through 6 we present the concurrency features of Hybrid and, based on our requirements, discuss their support for object-oriented programming and compare them to the approaches taken by other languages.

We conclude, in section 7, by underlining the features of Hybrid that offer satisfactory solutions as well as some promising ideas and open problems. Finally, in section 8, we present our future research directions concerning the further development of Hybrid as well as the development of a framework for the design of concurrency mechanisms suitable for object-oriented programming.

2. An Overview of Hybrid.

In a few words, Hybrid can be described as a strongly typed, concurrent object-oriented language supporting multiple inheritance, parameterization and persistence. In the next few paragraphs we give a short presentation of the features supported by Hybrid.

Hybrid is a strongly typed language. All objects in Hybrid are typed, with their type being specified by their visible interface (operations and instance variables). A subtype is a refinement of the parent type, achieved through the introduction of new (visible) operations and/or instance variables. Instances of subtypes can be used in place of instances of the parent type. Type checking in Hybrid is done statically at compile time, since all type information is known in advance. On the other hand, in order to enhance dynamic binding and support the introduction of new types at run time, Hybrid provides, in addition to static type checking, run-time type checking facilities.

Hybrid supports multiple inheritance. New classes can be created by inheriting from one or more existing classes. It should be noted that the inheritance hierarchy is independent of the type hierarchy.

Genericity is supported in Hybrid via the type parameterization mechanism. New types can be defined so that they make use of type parameters, which need to be bound at instantiation. The type parameters are constrained to be subtypes or type-equivalent to a specific type chosen by the programmer.

The persistence model of Hybrid is like the Smalltalk model, based on the workspace approach. All objects (and note, everything is an object in Hybrid) are stored in the persistent workspace. Execution can be interrupted at any moment and resumed at exactly the same point.

The Hybrid execution model is based on the notion of active objects. Every object in Hybrid may execute in parallel with other independent objects. Every independent “top-level” object, along with all its “sub-objects” (i.e. the objects that it contains and their sub-objects recursively), constitutes a domain. A domain is the unit of concurrency in Hybrid, in the sense that at most one of its objects will be active at any time. New domains can be dynamically created by instantiating and exporting objects of any type. A domain is identified by its object id (oid), assigned to it at creation. Object ids can be freely passed from one domain to another.

A domain can be in one of three states: idle, when no processing is taking place, running, when processing is taking place, and blocked, when waiting for the response to a call message.
Integrating Concurrency and Object-Oriented Programming

An Evaluation of Hybrid

Michael Papathomas
Dimitri Konstantas

Abstract

In this paper we address the effective use of the object-oriented programming approach for concurrent programming from a language design viewpoint. We present a set of requirements for the design of concurrent object-oriented languages. We then use a particular language, Hybrid, as a concrete example and examine to what extent its features meet these requirements. We identify the solutions offered by Hybrid and its shortcomings and we underline both the difficulties and promising directions for the design of concurrent object-oriented languages.

1. Introduction.

Apart from the more standard features supported by object-oriented languages[14], such as encapsulation and inheritance, there is a growing interest in the design of object-oriented languages that attempt to integrate a multitude of other features within the basic object-oriented framework, such as polymorphic type systems, concurrency, persistence and distribution. The integration of these features in object-oriented languages is promising for extending the application domain of object-oriented techniques.

In this paper we focus on the design of the concurrency mechanisms of object-oriented languages that successfully support the object-oriented approach for the development of concurrent software. In this respect, we formulate a list of requirements on the design of concurrent object-oriented languages which we then use for discussing the concurrency mechanisms of a particular language, Hybrid.

Hybrid [11] is an integrated object-oriented language and run-time system that was designed and implemented [8] at the University of Geneva. The aim of the project was to develop an object-oriented language that integrated a polymorphic type system, concurrency, persistence and distribution. Hybrid is well suited for the purpose of this paper since its design contains features that illustrate the difficulties in the use of object-oriented techniques for concurrent programming as well as satisfactory solutions that are valuable for the design of other languages with similar aims.

In section 2 we give a brief overview of Hybrid, focusing on its concurrency features. In section 3 we present a set of requirements for the design of the concurrency features of concurrent object-oriented languages. Our requirements are focused on the support for an object-oriented approach to concurrent programming that brings the discussion on the design of the concurrency features into the right perspective.