The ECO model: events + constraints + objects

G. Starovic, V. Cahill, B. Tangney
Department of Computer Science, Trinity College Dublin

Abstract

This document describes the rationale and design of a programming model based on events, constraints, and objects and the use of this model in the Moonlight[1] project. It describes the inter-object communication or invocation mechanism, and the way in which concurrency, synchronisation, and timing properties are expressed and controlled. The invocation mechanism is unusual in that it is *event-based*. It encourages loose coupling among the objects and this supports a high degree of encapsulation for each object. Concurrency, synchronisation, and timing properties are expressed in a uniform way using *constraints* which may be associated with objects and events. We describe the way in which the abstractions of the ECO model are expressed at the language level, and the support for them which is required from the runtime code and the underlying system.

1 Introduction

Large parallel and distributed applications are hard to program. Communication, synchronisation, and timing contribute to the complexity of this task. Object-orientation is advertised as a good paradigm for the modelling of entities in the application domain and a programming model which allows more structured and less complex program development. The Moonlight project is building an object-oriented environment for developing and executing games and virtual world applications. Some of the requirements coming from such an environment are:

- support different patterns of communication. As an example, a single object may collect information from a number of sources or disseminate information to a number of destinations. In general, there may be exchange of information between groups of objects, and the group membership may change dynamically.
- support soft real-time applications. It must be possible to express timing constraints on object behaviour. Such constraints arise out of the application domain and the way in which audio and video data are handled. When the constraints are occasionally not satisfied there are no catastrophic consequences for the system or for its environment.
- support distributed and persistent applications. An application may span a number of nodes in which case its objects communicate over a network. In some cases the objects will have to be persistent, i.e., retain their state across separate executions.
- support large applications with thousands of objects, where new objects may be created and the existing ones may disappear dynamically. This brings out the importance of issues like scalability and scoping rules.

This document describes the rationale and design of the ECO programming model and its use in the Moonlight project[2]. It includes the inter-object communication or invocation mechanism, and the way in which concurrency, synchronisation, and timing properties are expressed and controlled. A number of other important issues, like persistence, grouping, and mobility of objects are not considered in this document. The invocation mechanism is unusual in that it is *event-based*. It encourages loose coupling among the objects which supports a high degree of encapsulation for

[1] This work is partially funded by the CEC under ESPRIT contract No. 8636

[2] An earlier description of the same can be found in [13], which also describes other work on the Moonlight project done by the Distributed Systems Group at Trinity College Dublin. For more information on this project and work of all the partners involved contact moonlight@dsg.cs.tcd.ie.

each object. Concurrency, synchronisation, and timing properties are expressed in a uniform way using *constraints* which may be associated with objects and events. We describe the way in which the abstractions of the ECO model are expressed at the language level, and the support for them which is required from the runtime code and the underlying system.

The next section gives more details about objects with events and constraints and their possible implementation. Section 3 gives several examples and section 4 surveys some related work and compares it with the work reported in this document. The last section summarizes the main ideas, describes the state of the present implementation and sets out future work.

2 Objects, events, and constraints

The basic abstractions of the ECO model are objects, classes, events, and constraints. In this section we first briefly describe those properties of objects and classes which are relevant for the description of events and constraints.

Objects communicate by announcing events and by processing those events which have been announced. Each object is an instance of a class, it has instance variables and a number of methods which operate on these variables. A class specifies the interface to its instances (signatures of the methods which may be invoked on the instances), together with the events and constraints used by the instances. A method can be bound to one or more events in which case it behaves as an event handler. It is invoked when the event is announced, and it can itself announce one or more events. Several methods of an object can be bound to the same event. The type of an event determines the number and types of its parameters. In order to bind a method to an event the method signature has to match the event signature. The objects which announce an event are the *sources* of the event. Each occurrence of an event can affect zero or more objects (can be delivered to them causing invocations of their methods) — they are the *destinations* of the event. A source announces events without having to worry about the identities or locations of the destinations. Similarly, a destination object registers its interest in an event without having to worry about the objects which may announce the event. If necessary, both naming and location information can be expressed using event parameters.

Binding between a method and an event is dynamic. The method can stay bound to the event from the moment its object is created until the object is deleted. Alternatively, the method is bound at some arbitrary moment during the object lifetime and the binding can be changed after that. In our present design events have global scope, and sources and destinations may be located at different nodes of the distributed system. We intend to introduce some form of scoping at a later stage (possibly using the idea of spatial and temporal localities and area of interest managers described in [36]).

A constraint specifies a condition that should be either monitored-only or maintained and monitored. It is defined over some domain, in our case the domain includes event parameters, object instance variables, and possibly some constraint specific data. Constraints are evaluated at the observable points (the start and end of an event handler). The scope of a constraint is its enclosing class. There are different kinds of constraints, categorized by the data which they can access, by their evaluation points, and by the actions which they are allowed to perform. The information used by the constraints depends on the application. There may be a library of pre-defined constraints (e.g., those which implement typical synchronisation or timing constraints).

A program is a collection of cooperating objects, possibly placed on multiple nodes. When it is started, one of its objects must subscribe to the special *start* event announced by the system (a number of objects may subscribe to this event, i.e., there is not necessarily a single entry point per program). An ECO implementation[3] may automatically, or when instructed by the user, add a handler for this event to one or more objects and allow the user to override this default handler. The same can be done in some other cases, e.g., default handlers for special debugging events may

[3] A compiler or language preprocessor.

optionally be added to objects. Once the program is started the objects communicate with each other by announcing events and by being notified of event occurrences. They can also express their interest, or lack of interest in specific events. The program may decide to end when it learns about an occurrence of some event.

The ECO programming model can be made available in different existing languages. Two ways in which this can be done are ([7]):

- extend an existing language by making the new abstractions visible or explicit, or
- add support for the new abstractions using the existing language mechanisms (e.g., by inheriting from library classes which support the new abstractions).

Which approach is chosen depends on a specific language and the required extensions. The second approach may be easier to implement and easier to use (the original language remains unchanged). However, if the extensions are of a fundamental nature (e.g., a new inter-object communication mechanism, or a new form of inheritance), it may be difficult or impossible to integrate them seamlessly into an existing language. The first approach changes the language, with all the consequences which this brings (lack of compatibility with the old language, the extensions may not agree with the style of the original language). However, a language processor used by the first approach provides more flexibility, especially in the mentioned cases for which the second approach is less suited. In this section we show a way in which C++ [48] is extended with events and constraints.

2.1 Declaring events

Events have global scope and constraints have class scope. An event is defined with:

event EventName(parameters);

EventName is globally unique, and *parameters* is a list of event parameters (their names and types). A class declares its *in-events* and *out-events* with:

outevents list of EventNames;
inevents list of EventNames;

The former are those events which the instances of the class may announce, and the latter are those which they may handle. In a way, they are similar to the *import* and *export* statements in Modula-2 [53]. However, *in-events* and *out-events* differ from these statements. *in-events* lists those events in which the instances of the class may express interest at some moment during their lifetime. *out-events* lists those events which the instances may announce to their environment.

2.2 Notify constraints

Constraints are named conditions which use some data and which control the propagation and handling of events. A *Notify* constraint is optionally provided by a destination object when it subscribes to an event. The only data which can be used by this constraint are the values of event parameters, and the identity of the source[4] (plus optionally some constants). The destination object uses a Notify constraint to express: *I want to be informed about those occurrences of the event which satisfy this condition.* Since a Notify constraint does not depend on the local state of the destination object it can be evaluated in the context of a source object, or some *event manager* object. An example of a Notify constraint is given next.

constraint CountLevel { $(count = 1), (count + level < 2)$ }

CountLevel is the constraint's name, *count* and *level* are the names of two parameters[5]. of the event which is associated with this Notify constraint. The constraint requires

[4] It is assumed that each object has a unique identifier.

[5] *source* is used in a Notify constraint for the identity of the source object

that the value of the event parameter *count* is equal to 1, and that the sum of values of the event parameters *count* and *level* is less than 2.

A Notify constraint is associated with an event at subscribe time (when the destination object subscribes to the event). A group of objects may have mutual agreement that for example the first parameter of an event is the address of the intended destination object, or that it is the latest time when handling of a particular event occurrence should start, or that it is the priority of an event occurrence. Each of the destinations can use a different Notify constraint to specify when an occurrence of this event type qualifies to be delivered. This can be used to specify for example: *deliver to me those occurrences which are sent to me directly*, *deliver to me those occurrences which are sent with a sufficient maximum delivery delay*, or *deliver to me those occurrences which are sent with sufficiently high priority*. In a video game for example, a *collision manager* object may be used to detect collisions among game objects. It announces the *collision* event with the identities of the colliding objects passed as the event parameters. The interested objects may use Notify constraints as filters; only those collision notifications which are of interest to a specific object will be delivered to the object.

2.3 Pre and Post constraints

The *Pre* and *Post* constraints are used by a destination object as method wrappers. They use the object instance variables plus optionally constraint internal data, and may be used to implement:

- synchronisation within the object (e.g., Pre and Post constraints may be used to implement synchronisation variables from [19], these variables would be constraint internal data),
- control of the concurrency level within a method or within the object,
- timing control (e.g., earliest and latest method start-time and end-time, method duration from [3], and [33]),
- method pre- and post-conditions, method and object invariants — used for the runtime verification of object consistency and application correctness.

In addition to accessing and possibly modifying the instance variables and constraint data, Pre and Post constraints can announce an event, and Pre constraints can request that the current notification is: *discarded*, *enqueued*, or *processed*. This allows constraints to have *wait* or *failure* semantics [33]. In the case of *failure* semantics a constraint is used only to monitor a certain condition (e.g., the values of some instance variables). When a notification arrives and the pre-condition is not satisfied the constraint requests that the notification be discarded, optionally some event may be announced which will inform others about this failure. In the case of *wait* semantics when a notification arrives and it is found that the condition is not satisfied the Pre constraint may enqueue the notification for later processing[6]. Conceptually, each Pre constraint may have associated with it a queue of notifications. In order to allow the queued notification to be processed, a Pre or Post constraint may request *dequeuing* of a notification from one of the queues associated with the object's Pre constraints. When a notification is dequeued its Pre constraint will re-evaluate it, which may result in the notification being discarded, processed, or enqueued again. An example which shows how this works is given next.

A ResourceManager object manages some number of resources, and has one of its methods bound to the GetResource event and one of its methods bound to the FreeResource event (these events are announced by other objects). A Pre constraint for the method bound to GetResource can check if there are any available resources. If are none it

[6]This will be done when it is believed that the same notification may satisfy the condition at some later time, which may be the case for instance with synchronisation and timing constraints.

requests that the current event notification is *enqueued.* A Post constraint for the method bound to FreeResource requests that a notification is *dequeued* from the queue associated with the Pre constraint of the GetResource (if the queue is empty *dequeue* does nothing).

A Pre constraint may also request that a notification is *processed.* This is done when it is found that the condition is satisfied and that the object can proceed with handling the notification. There are two options: *process-active* and *process-passive* which can be used to control the level of concurrency within an object. If *process-passive* is requested there is a procedure call to the event handler (the event parameters are passed to the handler, which may require that they are unmarhsalled first if the notification is received from a remote source). If *process-active* is requested a new thread is created to execute the event handler (the event parameters are again passed to the handler). Each of the *discard*, *enqueue*, *process-passive* and *process-active*) statements ends the processing of the corresponding Pre constraint. Announcing an event and dequeuing notifications does not end the current constraint.

The *discard/enqueue/dequeue/process* options available to the constraints place the responsibility for implementing the synchronisation, timing, and other policies on the user. This mechanism has some potential disadvantages:

- it may be regarded as too low-level. However, this may not be a problem since we expect that there will be sets of frequently used constraints available to applications (e.g., constraints which implement one-writer/multiple-readers access policy, or which implement some typical timing constraints).
- The queueing of notifications may be too restrictive in some cases. There is a single queue per method, and the *enqueue* and *dequeue* allow appending to the end of the queue and removing from the front of the queue. Other possibilities (e.g., priority queues, various kinds of searching through the queue) may be required by some constraints. However, the described constraint options are intentionally left simple as it is expected that they will be sufficient for a number of applications[7]. In other cases, constraints may be implemented by specialised objects.

2.4 Announcing events and subscribing to events

An event is announced with:

announce EventName(parameters)

The *EventName* must be on the *out-event* list of the object's class. The announcement is asynchronous, the announcer does not wait for some "reply event" or for some object to handle the event. A method can be bound to an event initially (when the object is created), and can change its binding dynamically. The former is done in the class definition with:

MethodName(parameters) **handles** (EventName, NotifyName, PreName, PostName);

and the latter is done within the code with the *subscribe* and *unsubscribe* statements:

subscribe MethodName (EventName, NotifyName, PreName, PostName);

unsubscribe MethodName EventName;

in both cases the names of the constraints are optional. *MethodName* is local to the object which invokes *subscribe/unsubscribe*, and *unsubscribe* flushes the queue of the method/event Pre constraint. It is expected that *subscribe* and *unsubscribe* will be used to express object's current interest in certain events, while a Notify constraint will refine the specification of an object's interest in a specific event. It is possible to subscribe to or unsubscribe from a number of events. The following shows an example of a class with events and constraints:

[7] If required, it would be easy to increase the expressive power of constraints with extensions like: allow specification of *priority* with *enqueue* and *process-active*; or allow *flushing* of a queue.

```
event E1(···);
event E2(···);
event E3(···);
class myclass {
    inevents E1, E2;
    outevents E3;
    notify_constraints
        N { ··· }; // Notify constraint
    pre_constraints
        C1 { ··· }; // Pre constraint
    post_constraints
        C2 { ··· }; // Post constraint
    methods
        mymethod(···) handles (E1,N,C1,C2);
}

myclass::mymethod(···) {
        announce E3(···);
        unsubscribe mymethod E1;
        subscribe mymethod (E2,,,);
}
```

In this example, the method first subscribes to E1 with some constraints, and then (after announcing E3) it unsubscribes from E1, and subscribes to E2 without any constraints.

2.5 Implementation

This subsection describes a way in which some of the above concepts may be implemented, other implementations are possible.

Whenever an event definition is found in the code the event descriptor is registered in the Event Register (event descriptors are persistent and shared by the applications). Each of the events which appears on the *in-events* and *out-events* lists of a class must exist in the Event Register. In addition to this, for each event the code for marshalling and unmarshalling of its parameters has to be generated. At runtime, whenever an event is announced the information related to this event occurrence is used to evaluate the Notify constraints associated with the event. Event notifications are passed to the destinations of the satisfied Notify constraints. At the destination side, Pre and Post constraints are associated with methods, and support for *discard*, *enqueue*, *dequeue*, *process-active*, and *process-passive* is provided.

For the Notify constraints, there is code which will encode them and forward each of these constraints to all the sources of a specific event. At the source side, there is code which maintains the Notify constraints. The constraints are evaluated whenever their events are announced. An *event manager* object (EM) may be implemented per object/per event, per object (for all its events), per group of objects, or per node of the distributed system. One of the EM tasks can be maintaining and evaluating all the Notify constraints of the object's *out-events*. All the objects which can announce the same event can be registered as a group. If the underlying system supports group communication it can be used to inform all the sources about changes in the bindings (a new

Notify constraint added, or an existing Notify constraint removed). When an object is created, or when an existing object is brought into memory, it joins all the groups of its *out-events*. When an object is deleted, or moved out of memory, it leaves all the groups of its *out-events*.

Groups of event sources allow easier distribution of information about Notify constraints. In a distributed system it may be desirable to evaluate the Notify constraints as near the sources as possible, since this will stop the network traffic of unwanted notifications. An alternative to the groups of sources would be to use groups of destinations. This would make the distribution of notifications easier, but the possible price is distributing a lot of unwanted notifications if the Notify constraints are evaluated at the destination side. A third scenario which would have both: (a) groups of sources and Notify constraints evaluated at the source side and (b) groups of destinations, leaves open the question: "what criterion should be used to group the destinations". The conditions under which a notification is discarded by a Pre constraint application specific and with multiple such constraints it seems less likely that they can be used to form groups of destinations.

The use of the group communication mechanism described here is new. The usual way is to have groups of processes or threads, in our case there are groups of objects (it may be groups of EM objects). The only other reference that has groups of objects we know of is [30]. Also, groups are usually used for fault-tolerance, but as stated in [30] they can be used "as an addressing construct to accurately track a set of processes that share some characteristic". In our case we track sets of objects and the shared characteristic of the objects in a group is their ability to announce the same event. The group mechanism should be lightweight in order to cope with a large number of potentially overlapping groups [52]. The underlying system has to support lightweight threads and asynchronous communication (messages are used to communicate event occurrences to remote nodes). The basic requirement, with respect to the reliability and ordering properties of the underlying communication is: no guaranteed delivery and no guaranteed order. Some applications may require more, e.g., a causal or total order of the event announcements, subscribes, and unsubscribes.

3 Examples

It was already stated that events allow loose coupling between objects. An object may announce events for different reasons, some examples are:

- announce "x happened locally" (where x means a specific local action was performed or a specific local state was reached),
- announce "x happened locally, this will interest X", where X may be the name of some object or a group of objects. In this case the announcer knows the names of destinations,
- announce "I need y done by someone" (by anyone who can do it),
- announce "I need y done by Y" (where Y is the name of some object or a group of objects).

The first and third cases are anonymous communications, and second and fourth cases are named communications. With the event-based communication mechanism the names of destinations may be passed as event parameters, i.e., events support both anonymous and named communication.

The rest of this section shows different ways in which constraints can be used. The first example is of the previously described ResourceManager (slightly extended, the pool of managed resources may be *empty* or *full*). If a request for resource was announced and the pool is empty the request is queued; if a resource return was announced and the pool is full the return request is queued. In this example we assume that there is no need to control the level of concurrency within the object. The next example will show how this can be done. Also, the examples are sufficiently simple so that there is no need to use Notify constraints. Only the code related to constraints is shown.

```
class ResourceManager {
    pre_constraints
        PreGive { if (isempty) enqueue else process-passive; }
        PreRet { if (isfull) enqueue else process-passive; }
    post_constraints
        PostGive { if (wasfull) dequeue(PreRet); }
        PostRet { if (wasempty) dequeue(PreGive); }
    methods
        GiveResource(···) handles (GetResource,,PreGive,PostGive);
        ResourceReturned(···) handles (FreeResource,,PreRet,PostRet);
}
```

isfull, *isempty*, *wasfull*, and *wasempty* are boolean expressions which depend on the local state of the pool. The second example is of a consistent buffer. It manages some data and allows either multiple active *reads* or a single active *write* within the object:

```
class ConsistentBuffer {
    pre_constraints
        PreRead {
            if (current_write == 0) {
                current_read++;
                process-active }
            else enqueue; }
        PreWrite {
            if ((current_read == 0) && (current_write == 0)) {
                current_write++;
                process-active }
            else enqueue; }
    post_constraints
        PostRead {
            current_read- -;
            if (current_read == 0) dequeue(PreWrite); }
        PostWrite {
            current_write- -;
            dequeue(PreWrite);
            dequeue(PreRead); }
    methods
        Read(···) handles (ReadReq,,PreRead,PostRead);
        Write(···) handles (WriteReq,,PreWrite,PostWrite);
}
```

Dequeuing of a notification can be seen as causing an “internal object event”. The code which evaluates the object’s constraints is sequential, and such “internal events” are processed before processing of any external events is done. The level of concurrency is controlled at the observation points, it is not possible for a constraint or method to suspend or abort a method of the same object. Next, we describe the way in which some typical timing constraints can be implemented.

1. *start after time* and *start before time* requirements are implemented as Pre constraint. The *time* may be received as an event parameter or specified by the destination object. It may be required to enqueue a notification for later evaluation. In this case a timer event can be used to trigger dequeuing of such notifications and re-evaluation of the Pre constraints.

2. *finish after time* and *finish before time* requirements are implemented as either Pre or Post constraints. The *time* may again be received from the event announcer or specified locally. If the constraint is found to be unsatisfied an event may be announced which will cause error processing and possibly some recovery.

3. *maximum duration time* and *minimum duration time* are implemented with both Pre and Post constraints. Otherwise, they are similar to the above timing constraints.

In addition to synchronisation, concurrency, and timing, constraints can be used to express method pre-conditions, post-conditions, and invariants. Some of the ways in which they appear in other languages are given next (p is a boolean expression over the object state):

- *always p* or *invariant p*,
- *required p* or *when p*,
- *ensures p*.

The first case is a method invariant and it is implemented with both Pre and Post constraints. A method pre-condition (the second case) is implemented as a Pre constraint, and method post-condition (the last case) as a Post constraint. In these examples, if a Pre constraint is not satisfied the event notification is usually discarded (optionally some event may be announced). If a Post constraint is not satisfied it is usually accompanied by announcing some event.

4 Related work

A possibility of an event-based general-purpose communication mechanism has been suggested in [40]. This ought to be seen in the context of other proposals for language and system support for communication (where the communicating entities can be processes, threads, modules, or objects).

An early comparison of message passing and shared memory (or procedure-based) mechanisms is reported in [35] and [45]. Some of the more recent related work can be found in [1], [9], [21], and [52]. The remote procedure call (RPC) was introduced as a convenient extension of the procedure call [42]. Its basic form is synchronous, two-way, and one-to-one exchange of messages ([11], [17]). It encourages the client-server view of the world and influences the way in which programs are designed and implemented. The need for one-to-many, many-to-one, asynchronous, one-way, and other forms of communication has led to the extensions of the basic RPC ([8], [23], [51], [54]), and to completely different approaches (e.g., [12], [14], [16], [20]).

An event based language for parallel programming called EBL is described by Reuveni [44]. In this language events are the only control mechanism and cause the activation of event handlers. Event occurrences can be permanent or temporary and events can be recurrent or non-recurrent. Recurrent events can have multiple active occurrences, independently of whether they affect one or more destinations, and non-recurrent events can have only one active occurrence at any time (occurrences overwrite each other and only the last one survives).

The basic computational step is the announcement of an internal event (an event caused by the program, external events are caused by hardware). EBL is not object-oriented, instead a program consists of a collection of modules and each module consists of a number of event handlers. Events are typed; each event type has a name. All the occurrences of the same type of event have the same number and type of parameters (a parameter can be of an event type, in addition to simple types). The only action possible in an event handler is the announcement of one or more events. Several events can be announced sequentially or in parallel. A handler can be augmented with a condition which has to be satisfied before the handler is invoked. Reuveni also discusses the importance of scoping of events, the ways of achieving synchronisation with events, and the expressiveness of event based languages. Our work has been influenced by [44] and can be seen as an attempt to use some of these ideas in an environment which has objects and constraints.

The *generative communication* promoted by Linda [14] allows processes to communicate via the *tuple space*. A sender inserts a tuple (a list of typed data fields) into the space without having to worry about the identity and locality of the receivers. Receivers can inspect or remove tuples from this space by specifying a template tuple. The reception occurs when a match for the template tuple is found. Communication through tuple space is used in [38] in the context of distributed object-oriented languages. Oki et al. [43] use a variant of the Linda approach, called *anonymous communication*, where one field of each tuple is the subject field, and reception is based on the matching of the subject fields. Similar to the original approach, communication is independent of the identities and locations of senders and receivers. Agha and Callsen [2] describe Actorspace, a programming paradigm which integrates Actors [1] and Linda style communication. Actor-names can be expressions, they are evaluated in order to find the actors whose names satisfy the given expression. Actorspaces provide a scoping mechanism, are named and can form a hierarchy. The control of the names visibility, as well as control of the scope lifetime, is explicit and dynamic. Our approach has similar goals, but it is based on parameterized events and Notify constraints.

It is often stated that distributed systems require group communication, where the group membership changes and is determined by the global state of the computation (e.g., [2], [9]). Our work is in line with the attempts to support multiple and changing communication patterns. The loose coupling of objects avoids "the tendency of distributed naming systems to resolve names before communication occurs" (Bayerdorffer [9]), and our constraint mechanism allows communications to be specified in terms of local object states. The *associative broadcast* primitive of [9] allows the sender to provide an expression over attributes with each outgoing message. These expressions are evaluated locally where the potential receivers reside and depending on the outcome of this evaluation the messages are or are not delivered. Bayerdorffer considers events associated with naming and communication. Our events can be associated with naming and communication, but they can also be external events, timer events, and scheduling events [47].

Menon et al. [40] have thread-based and object-based event handlers. In ECO there are only object-based handlers. They also mention several applications for which events are especially suitable: distributed monitoring, debugging, and exception handling. The idea of loose coupling among communicating entities (this time to ease the integration of software components) is also used in [26] and [49]. There is insufficient space here to compare various other ways in which events are used (e.g., [22], [29], [34], [37], [46]).

Communication and control flow are often closely related — for instance communication primitives can be blocking or non-blocking. Depending on where and under what conditions this blocking is done it is possible to classify various primitives and languages with respect to their support for concurrency and synchronisation control [6]. There has been much work on language support for controlling the level of concurrency within objects and the order in which events occur. Arjomandi et al. [7] overview various approaches to adding concurrency support to a programming language. We use constraints to specify the level of concurrency within an object and do not make threads visible (except through *process-active* and *process-passive*). Some of the work on synchronisation constraints is reported in [10], [25], [39], and [50]. Frolund [25] have constraints specified as part of a class definition and each constraint restricts the set of methods which may be invoked when an incoming request is received. A constraint may depend on the parameters of the received invocation and the state of the target object[8]. Both [25] and [50] allow composition of constraints. The former is concerned more with the permissive and the latter with the restrictive aspect of constraints. In [25] each object has a controller which evaluates the constraints and may delay invocations (event deliveries) if there is a chance that this will make them acceptable in future.

The Archie language [10] allows specification of synchronisation states (or method pre-states), and method post-states, and integrates these states with type information. It also addresses the problem of multi-party synchronisation by introducing multioperations and coordinated calls based on [8]. In our case, the constraint mechanism can be used to express the required order of event announcements and deliveries at the level of a single object. Multiparty synchronisation may require complex expressions involving multiple events which we do not support at present. Our constraints allow the implementation of *activation conditions* [19], which are based on *synchronisation counters* [5]. An activation condition is attached to a method, and can depend on the instance variables, names of the methods, invocation parameters, and synchronisation counters. The counters are the object instance data maintained by the system and showing for instance the number of times each method was started, finished, or started and not finished.

The timing behaviour of a system is naturally described with constraints on event occurrences ([4], [18], [32]). Language support for expressing these constraints helps the development of programs which meet their timing specification [28]. Kenny and Lin [33] state that for a real-time system "there must be a way to define the constraints on time and resources to the computations. Some notion of a *constraint* must therefore be part of the system". Their language (Flex) has a constraint mechanism as a basic programming primitive. Flex constraints are associated with blocks of code. Exception handlers may be provided and will be executed when some of the constraints fail. An important concept used by various real-time languages is that of *observable points* [28]. They can be seen as markers, relevant for evaluating constraints, for making scheduling decisions and for tuning the code. Different languages have different notions of observable points. In our case, the observable points are at the object level (start and end of an event handler); in Flex they are at the level of a block of code.

The authors of [3] and [31] describe different ways of expressing timing constraints and integrating them into an object-oriented language. Timing behaviour can be described by specifying the minimum and maximum time when a certain observation point in the code is reached, or by specifying the time interval between two observation points. RTC++ [31] allows timing constraints both at the operation and statement level. It also allows a non-timing constraint to be specified for an operation, which can depend on the instance variables and message parameters. A function may be provided which is invoked when a constraint is not satisfied, and which will decide whether or not the invocation should be queued. The approach described in [3] relies on real-time *composition filters* for expressing timing constraints. There are input and output filters, specified at the class level. When an invocation message is received it is matched against the input filters for the class. The matching consists of evaluating a named expression which can depend on both instance and external variables. The method names can also be used for matching — a filter can be shared by several methods of an object. When a match is found the timing constraint from the corresponding filter is used. Our approach is similar but simpler (it has fewer basic abstractions) and more general.

Events and constraints have been used for constructing *active databases* with their Event-Condition-Action programming model (e.g., [15], [27]). Gehani et al. [27] support events and *triggers* in a database programming language. The events are of interest to one object or of interest to a group of objects and can be:

- *basic events* There is a number of predefined basic events, e.g., creation or deletion of an object, invocation of a member function, time-related and transaction-related events. A member function (its signature) can be used as a part of an event declaration.

[8]Frolund mentions the possibility of using "history instance variables" in the constraints.

- *logical events* They are the basic events optionally associated with *masks*. A mask is a predicate which specifies which occurrences of an event are of interest. It can use the parameters of the event being masked, or it can use the state of an object.
- *composite events, logical composite events* A composite event combines several logical events using the logical operators (and, or, not) and special event operators. The latter allow among other things specification of event order and periodic events.

The work reported in [27] is similar to our work in some ways. One important difference is that in our case events are used as a general communication mechanism. Also, we do not have composite events, but they can be supported at a higher level. A mask is similar to our constraint, but the latter cannot depend on the state of arbitrary objects. In [27] events are local to an object, and *triggers* are associated with a class definition. A trigger links an event with an action, and is active either perpetually or until the associated event is observed and the action is fired. The trigger corresponds to our facility to subscribe/unsubscribe to an event (both serve to link an event, constraint, and action). In [27] an action can be an arbitrary statement block while in our case an action is an event handler which is a method of some object.

In addition to being used for concurrency, synchronisation, and timing, constraints are used for specifying object invariants ([41]), and as a general construct in declarative languages (e.g., [24]).

5 Conclusions, present state and future work

This document describes the ECO programming model and its use in the Moonlight project. The event-based mechanism is used for communication among objects, it allows a higher degree of encapsulation and simplifies development of large and complex applications. A generalised constraint mechanism allows specification of a number of different requirements (synchronisation and concurrency within an object, timing behaviour of an object, and object's invariants). Although events and constraints have been used elsewhere, this combination of events, constraints, and objects allows a new and often more natural style of programming. Since events diminish the importance of object references they may allow new approaches to persistence and garbage collection.

At present, we are implementing the support for the ECO model in a single address space, which is the first requirement in the Moonlight project. In addition, the project aims at providing a set of tools which will help the user to create new games and virtual world applications. In such an environment, as already mentioned, there are a number of additional issues which will have to be resolved. One of them is scoping of events. Another is the required kind of inheritance. It is known that inheritance may interfere with synchronisation and timing constraints ([39], [25], [3]). In our case, constraints allow separation of the synchronisation and timing code from the "ordinary" application code. A library of typical constraints may be provided. It remains to be determined whether, in such an environment, there is a need for inheriting constraints and if there is then how it should be done. More important than this is to provide some support for expressing complex constraints which involve multiple events and multiple objects[9]. It may be possible to do this at a higher level using the basic building blocks described here.

Acknowledgements

The following people contributed with their comments and in other ways to the work reported here: A. Condon, A. Donnelly, N. Harris, S. McGerty, C. McHale, K. O'Connell, P. Taylor (Trinity College Dublin), D. Belsnes, B. Hauksson, A. Lie (Norsk Regnesentral Oslo), P. Caselles, L. del Pino (APD S.A. Madrid).

[9]A simple example of this is synchronous communication which involves ordered *request* and *reply* events, and may involve two or more objects.

References

[1] G. Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, 1986.

[2] G. Agha and C.J. Callsen. Actorspaces: A model for scalable heterogenous computing. Technical Report UIUCDCS-R-92-1766 and UILU-ENG-92-1746, Department of Computer Science, University of Illinois at Urbana-Champaign, November 1992.

[3] M. Aksit, J. Bosch, W. van der Sterren, and L. Bergmans. Real-time specification inheritance anomalies and real-time filters. In *ECOOP*, pages 386–407, July 1994.

[4] T. Amon. *Specification, simulation, and verification of timing behaviour.* PhD thesis, University of Washington, 1993.

[5] F. Andre, D. Herman, and J.P. Verjus. *Synchronisation of Parallel Programs.* North Oxford Academic, Oxford, 1985.

[6] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 115(1):3–43, March 1983.

[7] E. Arjomandi, W. O'Farrell, and I. Kalas. Concurrency support for C++: an overview. Technical Report CS-93-03, York University, Canada, August 1993.

[8] J-P. Banatre, M. Banatre, and F. Ployette. The concept of Multi-function: a general structuring tool for distributed operating system. In *Proc. of the 6th IEEE Distributed Computing Conference*, pages 478–485, 1986.

[9] B.C. Bayerdorffer. *Associative Broadcast and the Communication Semantics of Naming in Concurrent Systems.* PhD thesis, The University of Texas at Austin, December 1993.

[10] M. Benveniste and V. Issarny. Concurrent programming notations in the object-oriented language Archie. Technical Report 1882, INRIA-Rennes, December 1992.

[11] A.D. Birell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[12] K. Birman and R. Van Renesse. *Reliable Distributed Computing using the ISIS toolkit.* IEEE Press, 1993.

[13] V. Cahill, A. Condon, G. Starovic, and B. Tangney. Moonlight: VOID shell and execution environment definition. Deliverable 1.2.1. and 1.3.1, September 1994.

[14] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[15] S. Chakravarthy and D. Mishra. Snoop: an expressive event specification language for active databases. Technical Report UF-CIS-TR-93-007, University of Florida, Computer and Information Sciences, March 1993.

[16] A.T. Chandramohan, H.M. Levy, and E.D. Lazowska. Separating data and control transfer in distributed operating systems. Technical Report 94-07-04, Department of Computer Science and Engineering, University of Washington, July 1994.

[17] J.R. Corbin. *The art of distributed applications. Programming techniques for remote procedure calls.* Springer Verlag, 1991.

[18] B. Dascarathy. Timing constraints of real-time systems: constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, SE-11(1):80–86, January 1985.

[19] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill, and X.R. de Pina. A synchronisation mechanism for typed objects in a distributed system. In *OOPSLA*, pages 105–107, 1988.

[20] C.A. DellaFera, M.W. Eichin, R.S. French, D.C. Jedlinsky, J.T. Kohl, and W.E. Sommerfeld. The Zephyr notification service. In *USENIX*, Dallas, Texas, February 1988.

[21] M. Diaz, C. Chassot, A. Lozes, and K. Drira. On the space of multimedia connections. In *Cabernet Workshop, Trinity College Dublin*, January 1994.

[22] M. Donner, D. Jameson, and W. Moran. Events: a structuring mechanism for a real-time runtime system. In *Proc. of the Real-Time Systems Symposium*, pages 22–30, December 1989.

[23] N. Francez. Cooperating proofs for distributed programs with multiparty interactions. *Information Processing Letters*, 32:235–242, September 1989.

[24] B.N. Freeman-Benson and A. Borning. Integrating constraints with an object oriented language. In *ECOOP*, pages 268–286, June 1992.

[25] S. Frolund. Inheritance of synchronisation constraints in concurrent object oriented programming. In *ECOOP*, pages 185–196, June 1992.

[26] D. Garlan and D. Notkin. Formalising design spaces: implicit invocation mechanism. In *Lecture Notes in Computer Science 551: VDM Formal Software Development Methods*, pages 31–44, 1991.

[27] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 81–90, San Diego, California, June 1992.

[28] R. Gerber and S. Hong. Semantics-based compiler transformations for enhanced schedulability. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 232–242. IEEE Computer Society Press, December 1993.

[29] Object Management Group. Object services architecture, August 1992.

[30] O. Hagsand, H. Herzog, K. Birman, and R. Cooper. Object-oriented reliable distributed computing. In *I-WOOS*, 1992.

[31] Y. Ishikawa, H. Tokuda, and C.W. Mercer. Object-oriented real-time language design: constructs for timing constraints. In *ECOOP/OOPSLA*, pages 289–298, October 1990.

[32] F. Jahanian, R. Rajkumar, and S. Raju. Runtime monitoring of timing constraints in distributed real-time systems. Technical Report CSE-TR 212-94, University of Michigan, April 1994.

[33] K.B. Kenny and K. Lin. Building flexible real-time systems using the Flex language. *IEEE Computer*, 24(5):70–78, May 1991.

[34] T. Larrabee and C.L. Mitchell. Gambit: a prototyping approach to video game design. *IEEE Software*, 1(4):28–36, October 1984.

[35] H.C. Lauer and R.M. Needham. On the duality of operating systems structures. *ACM Operating Systems Review*, 13(2):3–19, April 1979.

[36] M.R. Macedonia, M.J. Zyda, D.R. Pratt, P.T. Barham, and S. Zeswitz. Npsnet: A network software architecture for large scale virtual environments. *Presence*, 3(4), 1994.

[37] N. Mansfield. *X Window System. A user's guide.* Addison Wesley, 1991.

[38] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. *SIGPLAN Notices*, 23(11):276–284, 1988.

[39] S. Matsuoka and K. Wakita. Synchronisation constraints with inheritance: what is not possible — so what is? Technical Report 90-010, Department of Information Science, The University of Tokyo, 1990.

[40] S. Menon, P. Dasgupta, and R.J. LeBlanc. Asynchronous event handling in distributed object-based systems. In *Proc. the 13th Conference on Distributed Computing Systems*, pages 383–390, Pittsburgh, Pennsylvania, May 1993.

[41] B. Meyer. *Eiffel: The Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[42] B.J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, 1981.

[43] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - an architecture for extensible distributed systems. In *ACM Symposium on Principles of Operating Systems*, pages 58–68, 1993.

[44] A. Reuveni. *The Event Based Language and its Multiple Processor Implementations*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1980.

[45] M.L. Scott. Messages vs. remote procedures is a false dichotomy. *SIGPLAN Noticies*, 18(3):57–62, May 1983.

[46] Y-P. Shan. An event driven Model-View-Controller framework for Smalltalk. In *OOPSLA*, pages 347–352, October 1989.

[47] G. Starovic. Scheduling and communication with events (unpublished internal document), June 1994.

[48] B. Stroustrup. *The C++ Programming Language. 2nd edition*. Addison-Wesley, 1991.

[49] K.J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[50] C. Tomlinson and V. Singh. Inheritance and synchronisation with enabled-sets. In *OOPSLA*, pages 103–111, October 1989.

[51] USL. Tuxedo system, release 4.2 manual, 1992.

[52] R. van Renesse, T.M. Hickey, and K.P. Birman. Design and performance of Horus: a lightweight group communication system. Technical Report 94-1442, Department of Computer Science, Cornell University, August 1994.

[53] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1982.

[54] M.D. Wood. Replicated RPC using Amoeba closed group communication. In *Proc. of the 13th Conf. on Distributed Computing Systems*, pages 499–507, May 1993.