An Event Based Object Model for Distributed Programming

Gradimir Starovic, Vinny Cahill, and Brendan Tangney

Distributed Systems Group,
Department of Computer Science,
Trinity College, Dublin 2,
Ireland.

E-mail: {Gradimir.Starovic,Vinny.Cahill,Brendan.Tangney}@dsg.cs.tcd.ie
URL: http://www.dsg.cs.tcd.ie/
Fax: +353-1-6772204

ABSTRACT: *This paper describes the rationale and design of a new distributed systems programming model based on events, constraints, and objects*[1]*. The paper 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 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.*

KEY WORDS: *events, constraints, inter-object communication mechanism, object-oriented programming*

1 Introduction

It is well known that large parallel and distributed applications are hard to program, [39], with communication, synchronisation, and timing contributing to the complexity of the task. It is also accepted that object-orientation goes somewhere towards simplifying these problems.

The application domain which we are currently addressing is that of distributed virtual reality especially next generation video games. 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.

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

- 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 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.

In order to meet these and other requirements a new programming model, known as ECO (for Events, Constraints and Objects) has been developed. This paper describes the rationale and design of the ECO model.

The invocation mechanism is unusual in that it is *event-based*. It encourages loose coupling among objects which 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. A number of other important issues, like persistence, grouping, and mobility of objects are not considered in this paper.

The next section gives more details about objects with events and constraints. Section 3 gives several examples and section 4 surveys some related work and compares it with the work reported in this paper. The latter section presents the ideas which influenced our work, and in a sense provides a justification for the kind of programming model described in this paper. 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 before going on to describe these items in detail.

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 together with the events and constraints used by the instances. Objects communicate by announcing events and by processing those events which have been announced. A method can be bound to one or more events and several methods of an object can be bound to the same event. A bound method behaves as an event handler in that it is invoked when the corresponding event is announced. A method can itself announce one or more events.

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. Thus the *sending* and *receiving* object are decoupled. However, 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 [32]).

A constraint specifies a condition which controls the propagation of events. For example an object can register interest in an event with the constraint that it should not be notified of occurrences if the value of the As described later 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.

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[2] 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 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 following sections go on to describe the model in more detail.

2.1 Language support

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, when the extensions are of a fundamental nature (e.g., a new inter-object communication mechanism), it may be difficult to integrate them seamlessly into an existing language. The first approach changes the language, with all the consequences which this brings (including lack of compatibility with the old language) However, a language pre-processor used with the first approach provides more flexibility.

The description in the following sections show how C++ [37] may extended with events and constraints.

2.2 Declaring events

Events have global scope and constraints have class scope. Objects can **subscribe** interest in or **raise** events. An event is defined with:

event EventName(parameters);

[2]A compiler or language preprocessor.

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 [41]. which the instances may announce to their environment.

2.3 Notify constraints

Constraints are named conditions 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[3] (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 other object (e.g. some special *event manager* object).

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.

2.4 Pre and Post constraints

The *Pre* and *Post* constraints are attached to the method/event bindings and behave as a destination object method wrappers. These constraints use the object instance variables and thus must be evaluated local to the object. They can also use constraint internal data. These type of constraints can be used to implement:

- synchronisation within the object (e.g., Pre and Post constraints may be used to implement synchronisation variables from [15]. ;, 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 [25]),

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

- 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 events, and Pre constraints can request that the current notification is: *discarded*, *enqueued*, or *processed*. This allows constraints to have *wait* or *failure* semantics [25].

For example consider a ResourceManager object which manages a 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 there are none, it 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 which can be used to control the level of concurrency within an object: *process-active* and *process-passive*. 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.

The *discard/enqueue/dequeue/process* options available to the constraints place the responsibility for implementing the synchronisation, timing, and other policies on the user. ;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[4]. In other cases, constraints may be implemented by specialised objects.

2.5 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);

[4]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.

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:

```
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.

3 Examples

As already described 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 a 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 constraints. 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.

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

The possibility of an event-based general-purpose communication mechanism has been suggested in [30]. 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 event based language for parallel programming called EBL is described by Reuveni [34]. 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 [34] 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 [12] 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 [28] in the context of distributed object-oriented languages. Oki et al. [33] 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 [36].

Menon et al. [30] 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 [19] and [38]. There is insufficient space here to compare various other ways in which events are used (e.g., [16], [22], [26], [27], [35]).

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], [18], [29], and [40]. Frolund [18] 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[5]. Both [18] and [40] allow composition of constraints. The former is concerned more with the permissive and the latter with the restrictive aspect of constraints. In [18] 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* [15], 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

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

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], [14], [24]). Language support for expressing these constraints helps the development of programs which meet their timing specification [21]. Kenny and Lin [25] 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* [21]. 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 [23] 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++ [23] 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., [13], [20]). Gehani et al. [20] 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.

- *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 [20] 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 [20] 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 [20] 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 ([31]), and as a general construct in declarative languages (e.g., [17]).

5 Conclusions, present state and future work

This paper has introduced the ECO programming model. An event-based mechanism is used for communication among objects as it allows a high 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 invariants). Although events and constraints have been used elsewhere, this combination of events, constraints, and objects allows a simpler and often more natural style of programming.

At present, we are implementing the support for the ECO model in a single address space. A library which maintains information about classes, objects, events, constraints, and various bindings is linked with and used by the application code. The library provides support for a number of predefined events and allows users to create and use new events. The library itself internally uses events. More details about this implementation can be found in [11].

Outstanding items to be addressed include an efficient distributed implementation and the type of inheritance to support. In the distributed implementation one of the key issues is scoping of events. While with inheritance it is known that inheritance may interfere with synchronisation and timing constraints ([29], [18], [3]). In our case, constraints allow separation of the synchronisation and timing code from the "ordinary" application code. 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. Also important is to provide some support for expressing complex constraints which involve multiple events and multiple objects[6]. 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: Andrew Condon, Alexis Donnelly, Neville Harris, Stephen McGerty, Ciaran McHale, Karl O'Connell, Paul Taylor (Trinity College Dublin), Dag Belsnes, Brynjulv Hauksson, Anund Lie (Norsk Regnesentral Oslo), Pascual Caselles, Luis del Pino (APD S.A. Madrid).

[6]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] V. Cahill, A. Condon, D.Kelly, S.McGerty, K. O'Connell, G. Starovic, and B. Tangney. Moonlight: VOID shell specification. Deliverable 1.5.1., March 1995.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

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

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

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

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

[29] 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.

[30] 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.

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

[32] Karl O'Connell, Vinny Cahill, Andrew Condon, Stephen McGerty, Gradimir Starovic, and Brendan Tangney. The VOID shell: A toolkit for the development of distributed video games and virtual worlds. In *To appear in the Proceedings of the Workshop on Simulation and Interaction in Virtual Environments*, 1995.

[33] 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.

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

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

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

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

[38] 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.

[39] Brendan Tangney, Andrew Condon, Vinny Cahill, and Neville Harris. Requirements for parallel programming in object-oriented distributed systems. *The Computer Journal*, 37(6), August 1994. Also technical report TCD-CS-94-03, Dept. of Computer Science, Trinity College Dublin.

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

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