Control-Driven Coordination Based Assembling of Components

Avraam Chimaris and George A. Papadopoulos

Department of Computer Science
University of Cyprus
75 Kallipoleos Str, P.O.B. 20537, CY-1678
Nicosia, CYPRUS

E-Mail: {cspgha,george}@cs.ucy.ac.cy

Abstract

The coordination paradigm has been used extensively as a mechanism for software composition and integration. Consequently, a number of associated models and languages have been proposed which address issues of CBSE from the coordination point of view. In this paper we use the control-driven approach to coordination and we present a framework within which it is possible to assemble components written in contemporary programming environments. In particular, we show how the coordination model IWIM, advocating a black-box approach to component development, can be realized in ActiveX. We map the fundamental building blocks of the IWIM model to ActiveX controls and we then show by means of an example the applicability of our approach, presenting also the results in a visual way, thus taking advantage of the features offered by the ActiveX environment.

Keywords**:** Coordination Languages and Models; System and Software Architectures; Distributed Software Composition; Reusability; Component-Based System Integration; Visual Programming.

1. Introduction

The concept of coordinating a number of activities, possibly created independently from each other, such that they can run concurrently in a parallel and/or distributed fashion has received wide attention and a number of coordination models and associated languages ([6]) have been developed for many application areas such as high-performance computing or distributed systems. In general, coordination models and languages fall into two main categories ([6]). The first one, we can call data-driven or shared dataspace approach. Its main characteristic is the use of a notionally shared medium via which the processes forming a computation communicate. The most notable realization of this approach is of course Linda ([1]). In Linda, the underlying view of the system to be coordinated (which is usually distributed and open) is that of an asynchronous ensemble formed by *agents* where the latter perform their activities independently from each other and coordination between them is achieved via some medium in an asynchronous manner. Linda introduces the so-called notion of *uncoupled communication* whereby the agents in question either insert to or retrieve from the shared medium the data to be exchanged between them. This shared dataspace is referred to as the *Tuple Space* and information exchange between agents via the Tuple Space is performed by posting and retrieving *tuples.* The shared dataspace approach, as expressed by Linda and the many other related coordination models ([6]), is " data-driven" in the sense that processes wishing to communicate with each other have to use primitives which retrieve or post data to the shared medium. This effectively means that the communicating components have to actually examine the sent or received data. The negative consequences of this approach are: (i) the coordination code of a program is intermixed with the computation code; (ii) it is difficult to re-use both the code as well as the general coordination pattern; (iii) as a result of these two reasons, the components are rather " glass boxes" (as opposed to being " black boxes") with whatever negative results this fact entails.

As a result of the above, another philosophy to developing coordination models and languages was proposed based on a "control-driven" approach ([6]). Contrary to what is happening in the shared dataspace approach to coordination, here processes communicate in a point-to-point manner by means of well-defined interfaces. Such a system evolves dynamically by means of raising and receiving control events. The coordinated components do not necessarily examine the data that is being transmitted through these point-to-point connections and therefore these components can be viewed as black boxes. This category, in its general form, encompasses many Software Architecture models, Architecture Description Languages (ADLs), Dynamic (Re-) Configuration Languages, etc. ([6]).

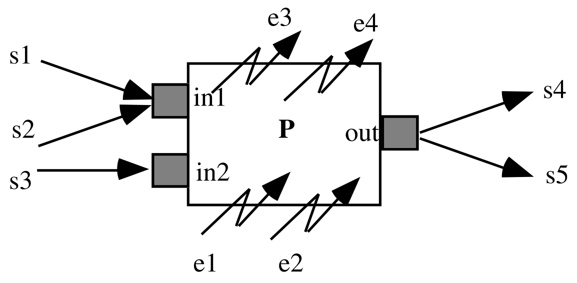
In this paper we show how one such model of control-driven coordination can be used as a " blue print" for developing component-based systems. In particular, we use the IWIM model ([2]) and we present a methodology whereby it is possible to build reusable components in any state-of-the-art programming environment such as ActiveX. The rest of the paper is therefore organized as follows: In the next section we describe the IWIM model; we then describe a general methodology of how the fundamental building blocks of IWIM can be realized in ActiveX. We show the applicability of our approach by means of an illustrative example, also taking advantage here of the visual capabilities of the ActiveX environment. The paper ends with some conclusions and reference to future work.

2. The IWIM Coordination Model

The Ideal Worker Ideal Manager (IWIM) model ([2]) is a control-driven coordination framework, as the latter has been defined above. In IWIM there exist two different types of processes: *managers* (or *coordinators*) and *workers*. A manager is responsible for setting up and taking care of the communication needs of the group of worker processes it controls (non-exclusively). A worker on the other hand is completely unaware of who (if anyone) needs the results it computes or from where it itself receives the data to process. Manager processes are written in Manifold, which is a concrete realization of the IWIM model (see below), whereas worker processes may be written also in Manifold or in some computational language (typically C, Fortran). In this latest case, these worker processes are called *atomics*. In particular, IWIM possesses the following characteristics:

- *Processes*. A process is a *black box* with well-defined *ports* of connection through which it exchanges *units* of information with the rest of the world.
- *Ports*. These are named openings in the boundary walls of a process through which units of information are exchanged using standard I/O type primitives analogous to read and write. Without loss of generality, we assume that each port is used for the exchange of information in only one direction: either into (*input* port) or out of (*output* port) a process. We use the notation `p.i` to refer to the port `i` of a process instance `p`.
- *Streams* or *channels*. These are the means by which interconnections between the ports of processes are realised. A stream connects a producer process to a consumer process. We write `p.o -> q.i` to denote a stream connecting the port `o` of a producer process `p` to the port `i` of a consumer process `q`.
- *Events*. Independent of channels, there is also an event mechanism for information exchange. Events are broadcast by their sources in the environment, yielding *event occurrences*. In principle, any process in the environment can pick up a broadcast event; in practice though, usually only a subset of the potential receivers is interested in an event occurrence. We write `e.p` to refer to the event `e` raised by a source `p`.

Activity in an IWIM configuration is *event driven.* A coordinator process waits to observe an occurrence of some specific event (usually raised by a worker process it coordinates) which triggers it to enter a certain *state* and perform some actions. These actions typically consist of setting up or breaking off connections of ports and channels. It then remains in that state until it observes the occurrence of some other event, which causes the *preemption* of the current state in favour of a new one corresponding to that event. Once an event has been raised, its source generally continues with its activities, while the event occurrence propagates through the environment independently and is observed (if at all) by the other processes according to each observer' s own sense of priorities.The figure below shows diagrammatically the infrastructure of an IWIM process.



The process `p` has two input ports (`in1`, `in2`) and an output one (`out`). Two input streams (`s1`, `s2`) are connected to `in1` and another one (`s3`) to `in2` delivering input data to `p`. Furthermore, `p` itself produces data, which via the `out` port are replicated to all outgoing streams (`s4`, `s5`). Finally, `p` observes the occurrence of the events `e1` and `e2` while it can itself raise the events `e3` and `e4`. Note that `p` need not know anything else about the environment within which it functions (i.e. who is sending it data, to whom it itself sends data, etc.).

As has been already mentioned, the IWIM model has been realised by means of a concrete coordination language, namely Manifold. However, the purpose of this paper is not to introduce this language; this is done extensively in many publications (see, for example, [4, 7]). Instead, we argue that IWIM is in fact independent of its concrete realisation and provides a complete —and tested! (due to the very existence of Manifold) —methodology for building component-based systems using any modern programming environment. Nevertheless, for illustrative purposes, but also for the reader to be able to follow and appreciate the work presented in the next section, we present below the Manifold version of a program computing the Fibonacci series.

```
manifold PrintUnits() import.
manifold variable(port in) import.
manifold sum(event)
  port in x.
  port in y.
  import.
event overflow.

auto process v0 is variable(0).
auto process v1 is variable(1).
auto process print is PrintUnits.
auto process sigma is sum(overflow).

manifold Main()
{
 begin: (v0->sigma.x, v1->sigma.y,v1->v0,sigma->v1,sigma-
>print).
 overflow.sigma:halt.
}
```

The above code defines `sigma` as an instance of some predefined process `sum` with two input ports (`x`,`y`) and a default output one. The main part of the program sets up the network where the initial values (`0`,`1`) are fed into the network by means of two “variables” (v0,v1). The continuous generation of the series is realised by feeding the output of sigma back to itself via v0 and v1. Note that in Manifold there are no variables (or constants for that matter) as such. A Manifold variable is a rather simple process that forwards whatever input it receives via its input port to all streams connected to its output port. A variable “assignment” is realised by feeding the contents of an output port into its input. Note also that computation will end when the event overflow is raised by sigma. Main will then get preempted from its begin state and make a transition to the overflow state and subsequently terminate by executing halt. Preemption of Main from its begin state causes the breaking of the stream connections; the processes involved in the network will then detect the breaking of their incoming streams and will also terminate. Please note that a visual presentation of this setup is shown in section 3.2.

In the example above note that both the computational process sigma and the coordination process Main, treat each other as black boxes: Main is only concerned with the input/output dependencies of sigma, and sigma operates completely unaware of Main’s doings. Note also that the actual data being produced and transmitted between the components of this apparatus (namely the Fibonacci numbers) do not play any role in the setup. Therefore, both Main and sigma are black boxes: Main is an Ideal Manager that can coordinate any computational process sigma assuming the above coordination pattern is reusable and applicable in other cases, and sigma is an Ideal Worker that will compute according to its specification without any knowledge of its surrounding environment. It is precisely these benefits that we want to convey to the development of component-based applications in the ActiveX environment (or any other such environment for that matter).

3. Realising IWIM concepts in ActiveX

In this section we show how the basic functionality of IWIM can be expressed in the ActiveX world. To that end, we created some general components (ActiveX and Classes) that support the IWIM functionality, simply and efficiently. These components are easily interconnected by the use of TCP/IP ports through which components can exchange messages. The main component control of our platform can raise an event, triggering an execution process that is linked on that event. These components are used as black boxes and the flow of the execution is separated into single steps on each control’s triggered event. Because in visual environments the main controller is the *form* object, we used the form object as the Manager of the model and a helping class for connecting and handling the other controls. Below we will try to analyse the main Controls and Classes that we used in our framework.

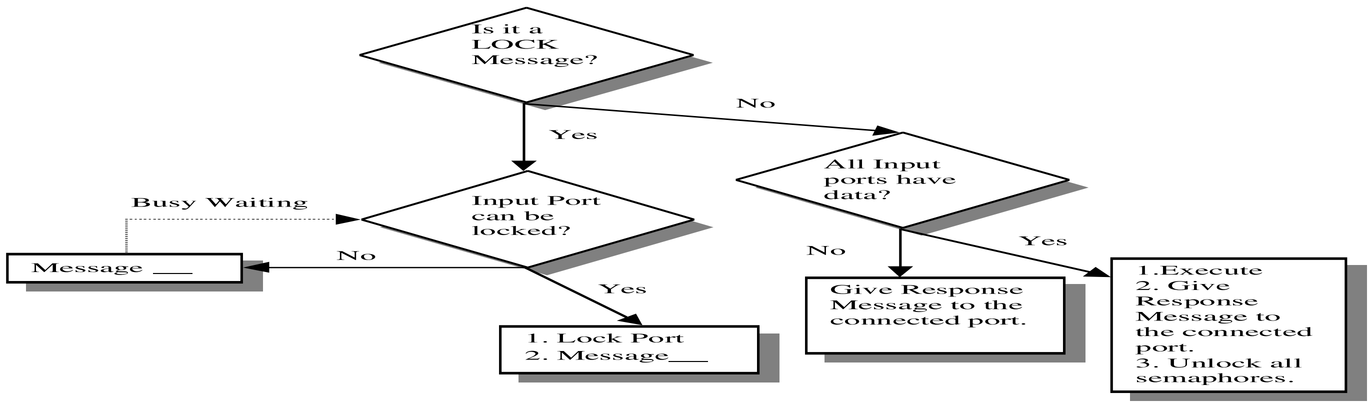
3.1. Analysis of the proposed approach

Realizing ports and port connections is one of the most important aspects of the IWIM model. We know that communication in IWIM is implemented by the use of one-direction ports (*input* port, *output* port). We use TCP/IP sockets in order to implement ports that could be able to exchange control and data signals between our components. In order to separate them by their role, we use the communication model of Client/Server to create logical channels between ports that would have the role of clients (output ports) and ports that would have the role of servers (input ports). We use the library CPSockets that contains the most important classes that compose the communication in a Client/Server fashion. This library includes Client/Server classes that can send and receive messages using a defined port. For class Client the most important properties are ServerHost and ServerPort that are used to define the Server that will accept and instantiate a communication channel. The main methods for creating and destroying the communication channel of the Client are Connect and CloseClient. The most important method of this class is RequestCustomResponse that is used for sending a stream message to a Server class. This method sends a given message and gets a reply from the Server that is connected to it. Easily with the use of the previous properties and methods, the Client class can be connected to a certain port and exchange messages by the use of method RequestCustomResponse. Similarly, the Server class has properties LocalPort and ServerName to determine the information of the Server that will be used from the Client for connecting to the class. Methods StartServer and StopServer are used for instantiating and destroying the communication channel. The OnCommand event is triggered when the Server is receiving a message from a Client class and a returned message is created to reply to the Client about the execution status. So, the Server is instantiated by using a given name and port. After the start method, the Server is able to accept connections and OnCommand events, and a reply message is sent to the connected Client.

The connectivity of the above classes is very simple. However they do not include any relative information for visual representation. Thus we implement two improved classes that contain Client class and Server class respectively, in order to assign information about their spatial position in the ActiveX space. The new classes are PORT_IN and PORT_OUT, which are using the properties XPoint and YPoint in order to define their position on their representation. Both classes have a Port property that is a *Client* type for the PORT_OUT class and a *Server* type for the PORT_IN class.

After we implemented the classes of communication, the next step was the creation of general visual components (ActiveX Controls). The components, due to their attributes and main functionality, can be separated into four categories:

- *Input Components*: responsible for the import of data from keyboard or files. They should use output ports to send data to other components.
- *Output Components*: responsible for the printing of data. They should use input ports to read data from other components and print them on an output unit (screen, printer, etc.).


Is it a LOCK Message?
No
Yes
All Input ports have data?
Busy Waiting
Input Port can be locked?
Message ___
No
No
Yes
Yes
1. Lock Port
2. Message___
Give Response Message to the connected port.
1.Execute
2. Give Response Message to the connected port.
3. Unlock all semaphores.

- *Process Components:* responsible for executing a computational process on entered data and forwarding results to other components. This type of components is the Ideal Worker in the IWIM model. These components use both input and output ports in order to read data from the connected neighbours and after processing forward results to certain recipients.
- *Managing Components:* managing connections of components and raising global events. This type of components is the Ideal Manager in the IWIM model.

The four categories above specify respectively the four general components that should be created for the implementation of our framework. Their attributes, methods and events were implemented in the controls: `INPUT`, `OUTPUT`, `WORKER` and `MANAGER` respectively. Below we give a general analysis of these controls.

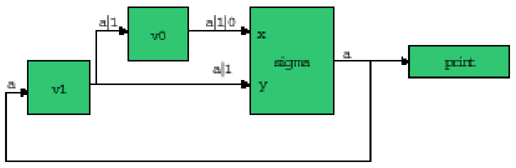
- **`INPUT`**: This component is in charge of importing data from keyboard or files, and sending this data to some other components. This component has a `P_OUTPUT` property, which is a `PORT_OUT` class, and uses a `READ_FILE` method to receive data from an input file. A double-click event is used to activate input from the keyboard. The `P_OUTPUT` is public in order to be handled by the `MANAGER` component. A `MANAGER` component is using the publicly defined ports to connect Client/Server Classes and also to visually connect ports by drawing a line. The colour of the line is changing from blue to red when this channel no longer exists.
- **`OUTPUT`**: This component is in charge of printing data. Data is printed in a multi-line RTF Control. This component has a `P_INPUT` property, which is a `PORT_IN` class, and prints its received data in the RTF Control.
- **`WORKER:`** This component is in charge of executing calculations on entry data and forwarding the processed data to other components. At the initialisation of the component, the developer must use a method (`SetPorts`), which defines the number of In/Out ports that will be used. The main method of calculation is executed after a certain event (the event `Comp_Step`). This event is raised automatically when the initialised entry ports contain data. So, every time new data arrives on input ports, is stored in variables till all ports contain data. At this point `Comp_Step` is raised, triggering the code in the event handler to execute and the variables are cleared for new data inputs. In order to avoid any loss of data between ports we use semaphores on input ports; in that way, we avoid overwriting input data in entry ports if the components have not as yet collected (i.e. removed from the ports) this data. The entry ports are continuously polled while there is data stored in them. Above is the diagram of the communication states of entry ports. When an output port is ready to send data, it checks if the connected to it input port of the other component is ready to receive data. Only if the semaphore of the port is unlocked a data stream is sent to the port. Otherwise the output port is waiting until the input port is unlocked after the connected control raises the step event (`Comp_Step`). In this way we have an asynchronous flow of streams in our connected diagram of components. Data packets are forwarded or delayed after a step event occurs or not. This control has four input ports (`P_INPUT1…4`) and four output ports (`P_OUTPUT1…4`). The main method of this control is the `SetPorts` method that is used to determine the input/output ports that will be used. We declare three methods for reading and sending data through the ports. Method `DATA_P_INPUT` is used for reading data that is recorded in a given entry port. This method is used in event `CompStep`, in order to make calculations on input data. The `SEND_TO_OUTPORT` method is used for sending data to a given output port. For avoiding fault executions of the `CompStep` event, we use a clock in each output port to make a small delay before raising the event. This methodology is changing the serial execution of events to a semi-threaded execution by using clocked events. `INITIALIZE_INPORT` is initialising an entry port with the initial values of the execution plan. Sometimes certain ports must start with a predefined data. Because we don' t have any variable classes as Manifold does, we created this method for initialising a given value on an input port. The last method is `DISABLE_COMP` witch is used for disabling the component from receiving new data in the entry ports. To call the process that will be executed, the control is raising the `CompStep` event.
- **`MANAGER`**: This Class is in charge of the networking of components. It includes the method `ConnectPorts` that connects input/output ports by assigning an available port number to the given ports. It also connects

graphically the components with lines. At the activation of this class, it detects the IP of the computer that is running the application, and initialises a socket port at a starting value. While new connections are created, the port number is increased in order to avoid any conflicts between communications among the involved ports.

At this point we should mention that in the first three visual components, the shapes that are assigned to the ports can change colour. Colour indicates if there is data stored on input ports or if an output port is waiting for an input port to get ready for new data. These coloured ports present a visual flow of data on the graph of connected components.

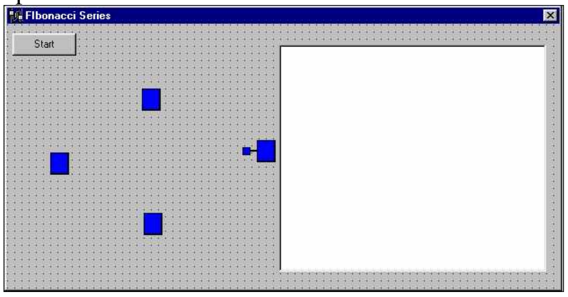
3.2. Modelling the Fibonacci Example in IWIM-compliant ActiveX

In this section we show how the Fibonacci example, introduced in the previous section, can be implemented in IWIM-compliant ActiveX. In order for the reader to appreciate the visual presentation that we were able to generate in the ActiveX environment, immediately below we show the setup that the involved processes create.



The above diagram shows that we need to create three WORKERs and an OUTPUT control in order to create the coordination diagram. Component v1 will be a WORKER control with one input port and two output ports, component v0 will be a WORKER control with one input port and one output port, and component sigma will be another WORKER control with two input ports and two output ports.

In our ActiveX form we will use a Manager class in order to connect the components between them. In the Design Mode the connections of ports are not presented. The Manager links at run time the ports between them and the interconnection lines are created. Below, is the form, at design time, in the Visual Basic environment with the various components. The button on the form contains the code for the initialisation of components v0 and v1. The OUTPUT component is a simple PRINT control with a single input port.



The equivalent to section 2 coordination code is shown below. The code of the class SUM is excluded; it simply adds a given pair of numbers. In our application we need an event that will denote to the MANAGER (form) the end of the application. We use a class (clsFibonacci) that is linked to our form (frmFibonacci) and contains the code that is executed by this event. Below is the code of our application:

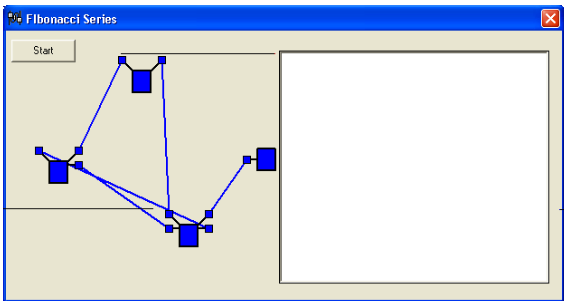
clsFibonacci

```
Private WithEvents mForm As frmFibonacci

Public Property Get Form() As Form
   'Reference to Form
   Set Form = mForm
End Property

Public Property Set Form(ByVal NewForm As Form)
   Set mForm = NewForm
End Property


Private Sub mForm_TooBig()
   'Code of Event
    MsgBox "The Limit was reached", vbInformation,
"Fibonacci"
End Sub
```

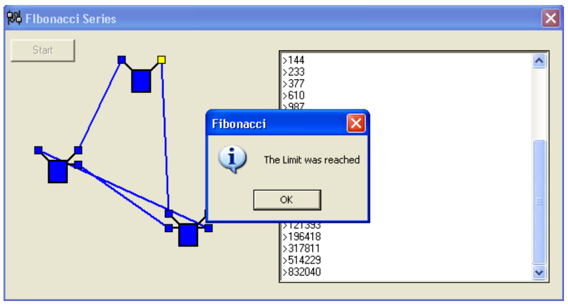
frmFibonacci

```
Option Explicit

Dim Manager As clsManager
Private cFibo As clsFibonacci
Event TooBig()

Private Sub cmdStart_Click()        'Initializing
Application
    Me.cmdStart0.Enabled = False
    Me.v0.INITIALIZE_INPORT 1, "0"  'Initializing v0
    Me.v1.INITIALIZE_INPORT 1, "1"  'Initializing v1
End Sub

Private Sub Form_Activate()
    Me.v1.SetPorts 1, 2   'Initializing Ports of v1
    Me.v0.SetPorts 1, 1   'Initializing Ports of v0
    Me.sigma.SetPorts 2, 2  'Initializing Ports of sigma
    With Manager
        .ConnectPorts Me, Me.v0, Me.v0.P_OUTPUT1, _
                      Me.sigma, Me.sigma.P_INPUT1
        .ConnectPorts Me, Me.v1, Me.v1.P_OUTPUT1, _
                      Me.v0, Me.v0.P_INPUT1
        .ConnectPorts Me, Me.v1, Me.v1.P_OUTPUT2, _
                      Me.sigma, Me.sigma.P_INPUT2
        .ConnectPorts Me, Me.sigma, Me.sigma.P_OUTPUT1, _
                      Me.print, Me.print.P_INPUT
        .ConnectPorts Me, Me.sigma, Me.sigma.P_OUTPUT2, _
                      Me.v1, Me.v1.P_INPUT1

'With the code above we connect the four components
'As we saw in the diagram above there are 5 connections.
    End With
End Sub

Private Sub Form_Load()            'Initialize and connect
Event Class
    Set Manager = New clsManager
    Set cFibo = New clsFibonacci
    Set cFibo.Form = Me
End Sub

Private Sub sigma_CompStep()       'sigma execution step
Dim dblFib As Double
    'Reading input ports, add values and sending to
output port.
    dblFib = CDbl(Me.sigma.DATA_P_INPUT(1)) + _
           CDbl(Me.sigma.DATA_P_INPUT(2))
    If dblFib > 1000000 Then    'Checking Limit (1000000)
        RaiseEvent TooBig 'Calling Event if limit reached
    Else
        Me.sigma.SEND_TO_OUTPORT 1, dblFib
        Me.sigma.SEND_TO_OUTPORT 2, dblFib
    End If
End Sub


Private Sub v0_CompStep()          'v0 execution step
    Me.v0.SEND_TO_OUTPORT 1, Me.v0.DATA_P_INPUT(1)
End Sub


Private Sub v1_CompStep()          'v1 execution step
    Me.v1.SEND_TO_OUTPORT 1, Me.v1.DATA_P_INPUT(1)
    Me.v1.SEND_TO_OUTPORT 2, Me.v1.DATA_P_INPUT(1)
End Sub
```

The state of the application at run-time, before the initialisation of `v0` and `v1`, is shown below in the first figure. Notice the similarity between the figure of our form and that of the diagram at the beginning of this subsection. After we press the `Start` button the components begin receiving and sending messages up to the final situation (second figure) where the Manager (form) receives the event



`TooBig.`



```
>144
>233
>377
>610
>987
>121393
>196418
>317811
>514229
>832040
```

The Limit was reached

4. Conclusions

In this paper we have presented a methodology for building component-based software, using the coordination paradigm. More to the point, we have shown how the IWIM model can be realized in a state-of-the-art programming environment, namely ActiveX. We have developed techniques, particular to the ActiveX environment, which model the basic building blocks of the IWIM model. We then showed how these can be used to implement a simple, but hopefully illustrative, example.

The realization of the IWIM model in other (than its own language Manifold) programming environments has been attempted twice (to our knowledge) before. In [5] we show how the basic Linda model can be adapted to function according to the IWIM philosophy, using a set of tuple communication protocols that resemble the point-to-point communication infrastructure of IWIM. In [3], the authors report of an adaptation of the IWIM model in Java, where point-to-point communication is realized by means of JavaPorts, a set of tools that they have developed for that purpose. It appears that their model, though, does not support the events functionality of IWIM and, consequently, it is unable to support dynamic evolution of a component setup. This is not the case of the approach described here or in [5].

The next step in our work is to move from the " blue print" to the building of an environment, which will automatically create IWIM compliant ActiveX components and assemble them together using a visual interface as the one presented in the third section. Interfacing those components with other components, also IWIM compliant and written in a variety of languages, is another interesting aspect of our future work.

5. References

[1] S. Ahuja, N. Carriero and D. Gelernter, " Linda and Friends" , *IEEE Computer* 19 (8), 1986, pp. 26-34.

[2] F. Arbab, " The IWIM Model for Coordination of Concurrent Activities" , *First International Conference on Coordination Models, Languages and Applications (Coordination' 96),* Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 34-56.

[3] E. S. Manolakos and D. G. Galatopoulos, " JavaPorts: An Environment to Facilitate Parallel Computing on a Heterogeneous Cluster of Workstations" , *Informatica*, Vol. 23 (1), April, 1999, pp. 97-105.

[4] G. A. Papadopoulos and F. Arbab, " Coordination of Systems With Real-Time Properties in Manifold" , *Twentieth Annual International Computer Software and Applications Conference (COMPSAC' 96),* Seoul, Korea, 19-23 Aug., 1996, IEEE Press. pp. 50-55.

[5] G. A. Papadopoulos and F. Arbab, " Coordination of Distributed Activities in the IWIM Model" , *International Journal of High Speed Computing*, World Scientific, 1997, Vol. 9 (2), pp. 127-160.

[6] G. A. Papadopoulos and F. Arbab, " Coordination Models and Languages" , *Advances in Computers*, Marvin V. Zelkowitz (ed.), Academic Press, Vol. 46, August, 1998, 329-400.

[7] G. A. Papadopoulos, " Distributed and Parallel Systems Engineering in Manifold" , *Parallel Computing*, Elsevier Science, special issue on Coordination, 1998, Vol. 24 (7), pp. 1107-1135.