REACTIVITY AND PRO-ACTIVENESS IN VIRTUAL PROTOTYPING

Paulo César Rodacki Gomes
Bruno Feijó
Renato Fontoura de Gusmão Cerqueira
Roberto Ierusalimschy
Department of Computing
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro, RJ, Brazil

ABSTRACT

Virtual Prototyping becomes much more interesting when 3D solids are dynamic distributed objects on heterogeneous platforms. This paper proposes the concept of Extended Constraint Graphs to represent reactivity and pro-activeness in virtual prototypes. Also, this paper presents an original approach to deal with the dynamic nature of virtual prototypes through the use of an interpreted language Lua and a CORBA binding called LuaOrb. Moreover, this work presents a CORBA model for virtual prototyping based on the geometry bus ACIS in which objects and attributes are created and modified at run time.

KEYWORDS

Collaborative Design Support, Solid Modeling, Distributed Computing, Virtual Prototype, Dynamic Invocation Interface.

1. INTRODUCTION

The aim of virtual prototyping is to build a full virtual artifact in such a way that design and manufacturing problems are anticipated and discussed within a cooperative work environment. The objects of the virtual artifact have several types of attributes, such as geometric attributes, design intent attributes, manufacturing attributes, cost attributes, part number references and documentation references. Furthermore, these objects should be reactive and pro-active, in the sense that they react to changes in the environment and trigger actions automatically. Reactivity and pro-activeness are in the kernel of the dynamic nature of the design processes. The problem is how to implement reactive and pro-active objects which are distributed on different networks. Figure 1 illustrates the basic problem to be solved. For instance, the object box2 may wish to scale down the radius of the remote cylinder by 2/3. However, this change should be done while the two applications are running simultaneously. The authors claim that the solution for this type of problem should consider geometry buses and object-distributed computing.

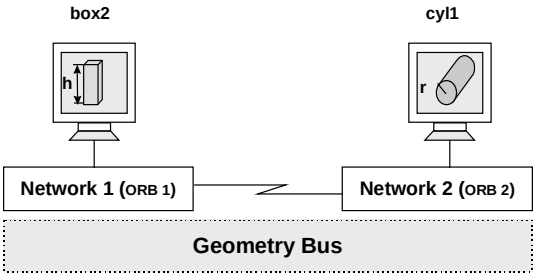

Figure 1. A Distributed Virtual prototype

A *Geometry Bus* consists of a standard object-oriented geometric library, a component architecture and a standard geometric data format, such as the ones found in ACIS (Spatial Technology , 1996). CORBA (Object Management Group, 1995) has been proposed as a distributed object architecture for integrated CAD systems (Regli, 1997). However, there are virtually no references in the literature on ACIS-CORBA integration and dynamic distributed objects in CAD systems. Also, there are few references to reactivity and pro-activeness in virtual prototyping in the literature (Feijó et al, 1996). This paper presents the concepts of reactivity and pro-activeness in virtual prototyping which can be implemented by a CORBA model based on ACIS geometry bus and dynamic distributed objects.

2. REACTIVITY, PRO-ACTIVENESS AND EXTENDED CONSTRAINT GRAPHS

Reactivity means that objects are able to perceive the environment and react to changes in this environment. Pro-activeness means that objects act oriented by their goals and trigger actions automatically.

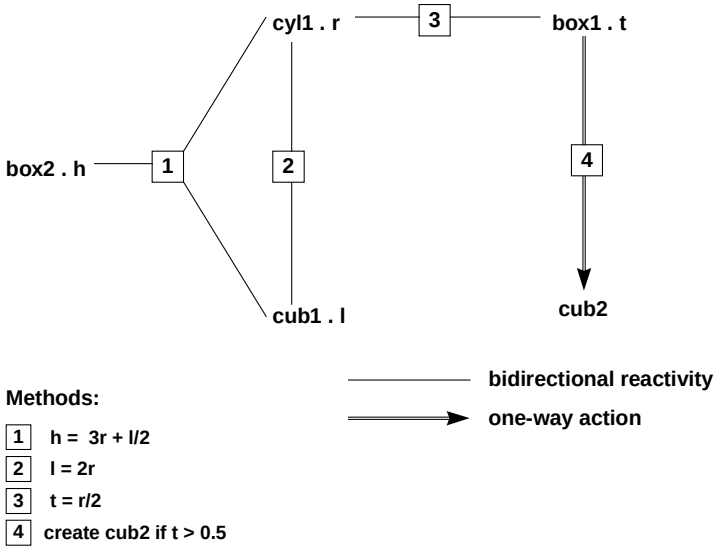

Figure 2 An Extented Constraint Graph

Pro-active objects are committed to other objects through direct actions which are constantly fired. Firing is inherent to the real-time nature of a society of dynamic distributed objects. This active behaviour is similar to those found in active database systems (Widom and Ceri, 1995).

Both reactivity and pro-activeness can be shaped in terms of agency. In this case, the question of computational intelligence arises and is not within the scope of the present paper (Bento and Feijó, 1997).

In order to illustrate the dynamic nature of virtual prototyping, one may imagine a virtual prototype made of four virtual 3D objects distributed in four different platforms but strongly committed to each other. In this virtual world, any change in a 3D environment triggers reactions in other environments. Also, but most important, any user or object may create a new 3D object or change relationships between objects at run time. For instance, the change of the radius of a cylinder *cyl1* in the virtual prototype may cause the object *box1* create a fifth object called *cub2*.

This paper proposes a special graph, called Extended Constraint Graph, to represent reactivity and pro-activeness in virtual prototypes. A possible Extended Constraint Graph for a virtual mock-up is shown in Figure 2. An Extended Constraint Graph is denoted as:

$G = (V, M, E)$

where V and M are sets of vertices representing the variables and methods, respectively, and E is a set of partially directed edges denoting the graph relationship between variables and methods. Variables have the form:

obj.attribute

Variables of the form

obj

denote the reference to an object. Methods can represent a *bidirectional reactivity* (e.g. a constraint equation), or a *one-way action* (e.g. method 4 in Figure 2).

In a design process, this type of graph is built and modified at run time by the user or an object. For instance, one may suppose that the initial state of the virtual mock-up is { r = 1, h = 4, l = 2, t = 0.5}. If the user changes the cylinder's radius to r = 2, the new state of the virtual mock-up will be { r = 2, h = 8, l = 4, t = 1, cub2 is created}. Alternatively, the user might have start changing the height of box2.

Methods for bidirectional reactivity are defined by three kinds of elements: *(1)* a condition (usually a constraint equation); *(2)* the inverse of this condition; and *(3)* a piece of code for each direction of the reactive link. For instance, the link between cyl1.r and box2.h is defined by the following conditions and pieces of pseudo-code:

direction cyl1.r - box2.h:

if *condition* $r = 1/3(h - l/2)$ is not satisfied, then add the value of $h = 3r + l/2$ to box2

direction box2.h - cyl1.r:

if *condition* $h = 3r + l/2$ is not satisfied, then add the value of $r = 1/3(h - l/2)$ to cyl1

The operation of adding a new value to an object should trigger other reactive links that this object might have. Figure 3 illustrates this bidirectional process for the case of someone making a change in the value of the cylinder's radius in accordance with the Extended Constraint Graph of Figure 2.

Reactivity and pro-activenesss represented by the Extended Constraint Graph can be implemented by *attached procedures* and *link-to* relations. Attached procedures are used to implement one-way actions. Attached procedures are employed for the purpose of triggering procedures on variable access, drawing on the style of active values and access-oriented programming techniques (Stefik, Bobrow and Khan, 1986) (Inference Corporation, 1985). There are two types of attached predicates: *if-needed* and *if-changed*. *If-needed* predicates are evaluated before an attribute value is retrieved. Oppositely, *if-changed* procedures are executed in the case of an attribute value being tentatively changed - once more, before assertion of the attribute value takes place. *If-changed* procedures behave like "watch-dogs" of the attributes to which they are attached. For instance, the attribute *t* of *box1* is defined with an *if-changed* procedure which establishes that the object *cub2* should be created if $t > 0.5$.

Link-to relations make bidirectional reactivity possible. Every object *i* linked to other objects has a list $link\text{-}to_i$ with members of the form:

$$< OBJ_k.t_n, R_{ij}^{kn} >$$

which denotes a link between the attribute t_j of the object i with the attribute t_n of the object k, that is:

$$OBJ_i.t_j \text{ ---------- } OBJ_k.t_n$$

where R_{ij}^{kn} is defined by the following function:

```
function r_ijkn (OBJi, OBJk)
   get attribute Tj of object OBJi
   get attribute Tn of object OBJk
   get other attributes AT1, AT2, ... of
     other objects, if any
   if condition(Tj, Tn, AT1, AT2, ...)
     is not satisfied, then
       compute the new value of Tn
         using the inverse of condition
       return add_value(Tn to OBJk)
   end
   return TRUE
end of function
```

Before adding a new value to an attribute of an

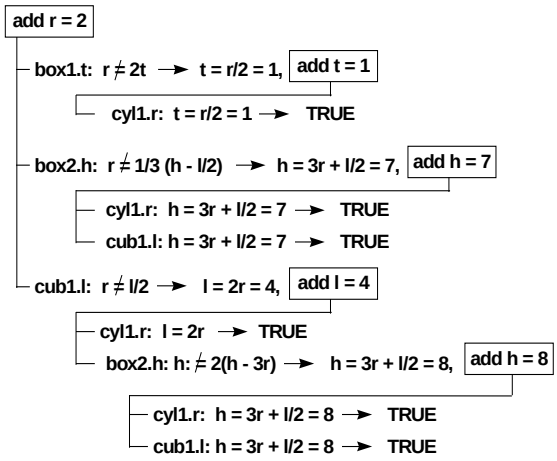

Figure 3 A Bidirectional Reactivity Process

object, the function `add_value` executes every function *r_ijkn* found in the *link-to* list of the object. This mechanism guarantees the full propagation of the changes in all directions.

In a design process, the Extended Constraint Graph of a virtual prototype is constantly changed at run time. This means that old vertices are deleted, new vertices are created, methods are revised and new methods are created by the user in the distributed environment on the network. In terms of CORBA, this dynamic behavior can only be achieved with the Dynamic Invocation Interface (DII). However, the current language bindings to CORBA are not flexible enough to respond effectively to the dynamic demands of the design process. Additionally, even with a more adequate language, there is still the problem of writing an ACIS component for this language. This paper addresses these two problems in the following sections.

3. ACIS

An informal discussion about ACIS and its possibilities of becoming a *de facto* standard can be found in Porter (1995). The official reference for ACIS is Spatial Technology (1996). A Geometry Bus can be provided by ACIS architecture, ACIS components (called Husks) and ACIS SAT file format. All products and

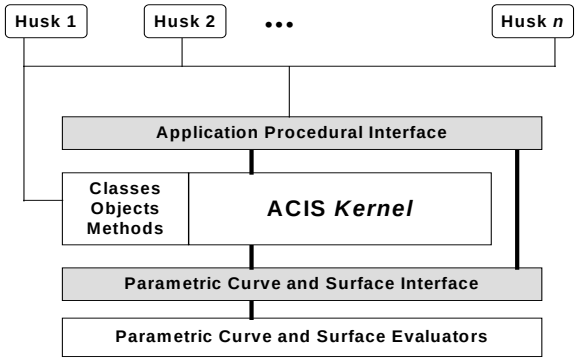

Figure 4. ACIS Architecture

applications built on ACIS support the ACIS SAT file format. Therefore, a CAD user can take a SAT file of a model and seamlessly transfer it back and forth among a large number of ACIS-enabled applications. At the present moment, most of the leaders in the low/medium-level of CAD systems have ACIS in their kernel, such as Corel CAD™, AutoCAD™, MicroStation™ and SolidEdge™.

The architecture of the ACIS Kernel Modeler is shown in Figure 4. ACIS Kernel is a Boundary-representation (B-rep) modeler that integrates 3D solids, surfaces and wireframes. A Geometry Bus requires the reuse of software components. ACIS meets this requirement through the use of component modules called *Husks*. In a lower level, the ACIS C++ class structure is another platform for the reuse of software components. Any of these components (lower-level modules or *Husks*) can be substituted by a newer component developed by the programmer without jeopardizing consistency and interoperability.

4. CORBA AND DYNAMIC INVOCATION INTERFACE

CORBA (Common Object Request Broker Architecture) is an architecture for interoperability between applications on different machines in heterogeneous distributed environments (Siegel, 1996) (Object Management Group, 1995). CORBA allows applications to communicate with one another no matter where they are located or in which language they are written. The CORBA proposal is strongly based on the notion of object interfaces, described in IDL (Interface Description Language), which enable client/server object interaction within a specific implementation of an Object Request Broker (ORB).

The ORB is the middleware that establishes the client/server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. Actually the ORB is able of searching for the object that can implement a specific request. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface.

ORBs let programmers choose the most appropriate operating system, execution environment and even programming language to use for each component of a system under construction. In an ORB-based solution, developers simply model the legacy component using the same IDL they use for creating new objects. In the case of legacy components, the programmer generates an IDL interface to represent the module's functions and codes a thin "wrapper" to connect the IDL layer to the legacy interface.

The development of mappings (also called bindings) from IDL to different programming languages allows each client and server to be implemented in the most convenient language. CORBA supports two types of client/server invocations: static and dynamic. Static invocations are stubs generated at compile time, from information in the IDL description of objects to be invoked. Dynamic invocations use the Dynamic Invocation Interface (DII), which allows a client to build and invoke requests to servers at run time.

A major difference between DII and static interfaces is that DII allows the discovery of the object type and methods at run time, while static interfaces require the definition of the object type to be made at compile time. In other words, DII

allows dynamic typing. Most CORBA language bindings, including the bindings for C, C++, Smalltalk, Java. are strongly based on the static invocation interface. Although they also support access to DII, this support is in a very low level of abstraction. Indeed, since most of these languages are statically typed (except Smalltak), it is inherently difficult to bind DII facilities to higher level language constructs. The Smalltalk biding makes use of the dynamic characteristics of the language (Goldberg and Robson, 1983) to generate stubs at run time, however the biding is still based on static IDL descriptions and stubs. TclDii is a non-statically typed language (Almasi et alii, 1997) built on top of DII. However, TclDii obliges the client programmer to know many details of DII, because the use of DII is not transparent in TclDii. Therefore, there is no CORBA biding appropriate to deal with the highly-dynamic nature of design processes in distributed object environments. A promising way towards an easy and flexible use of DII is the strategy of mapping the dynamic character of an interpreted, reflexive and dynamic typed language to the dynamic characteristics of the Dynamic Invocation Interface.

Cerqueira et al (1997) developed a binding between the extension language Lua (Ierusalimschy et al, 1996) and CORBA, called LuaOrb. The next two sections summarize the concepts of Lua and LuaOrb.

5. LUA

Interpreted languages are very useful for a number of cases, such as rapid prototyping in requirement acquisition, customizable applications and mutable, object-oriented applications. Nowadays, many programs are split in two parts, a kernel and a configuration, written in two different languages. The kernel implements the basic classes and objects of the system, and is usually written in a compiled, statically typed language, like C. The configuration part, usually written in an interpreted, flexible language, connects these classes and objects to give the final shape to the application (Ousterhout,J.K., 1997). Such configuration languages are also called extension languages.

Lua, an extension language, integrates in its design data-description facilities, reflexive facilities and familiar imperative constructs. On the "traditional" side, Lua is a procedural language with usual control structures (*while, if*, …), function definitions with parameters and local variables. On the less traditional side, Lua provides functions as first order values, and dynamically created associative arrays (called *tables* in Lua) as a single, unifying data-structuring mechanism.

As a simple example of Lua code, Figure 5 shows the implementation of the map function in Lua. A map receives a list `a=(a1,a2, ...,an)` represented as a table, and returns a new list `b` in which the original elements are transformed by a specific function `f`, i.e. `b=(f(a1),f(a2), ...,f(an))`.

There is no notion of a "main" program in Lua; being an embedded language, it only works embedded in a host language. Lua is provided as a library of C functions to be linked to host applications. The host can invoke functions in the library to execute a piece of code in Lua, write and read Lua variables, and register C functions to be called by Lua code. Moreover, *fallbacks* can be specified to be called whenever Lua does not know how to proceed.

```
function map (a, f)
   local b = {}        -- creates a new table
   local i = 1
   while a[i] do
      b[i] = f(a[i])
      i = i+1
   end
   return b
end
```

Figure 5. An Example of Lua Code

Functions in Lua are first class values. A function definition creates a value of type function, and assigns this value to a global variable (e.g. `map`, in Figure 5). Like any other value, function values can be stored in variables, passed as arguments to other functions, or returned as results. This feature greatly simplifies the implementation of object-oriented facilities.

Lua is dynamically typed. Variables can handle values of any type. Whenever an operation is performed, it checks the correctness of its argument types. Besides the basic types *number* (*floats*) and *string*, and the type *function*, Lua provides three other types: *nil*, *userdata*, and *table*. The type *userdata* is provided to allow arbitrary host data to be stored in Lua variables.

Typically, userdata values are used to store C pointers created by C functions, which can be passed back to other C functions, behaving as an abstract data type in Lua.

The type *table* implements associative arrays, that is, arrays that can be indexed not only by integers, but by strings, reals, tables, and function values. Tables can also implement records by simply using field names as indices. Lua supports this representation by providing `a.name` as syntactic sugar for `a["name"]`.

Tables are created with special expressions, called *constructors*. The simplest constructor is the expression {}, which returns a new empty table. An expression like *{n1 = exp1, n2 = exp2, …}* creates a new table, and stores in each field *ni* the result of *expi*. A typical example is the creation of a point: `point1 = {x = 10, y = 30}`. Constructors can also be used to built lists, with the syntax *{exp1, exp2, …}*, such as `days = {"Sunday", "Monday"}`.

In Lua, method definitions can be written as:

```
function object:foo(a1,a2,...)
   ...
end
```

which is equivalent to:

```
function object.foo(self, a1,a2,...)
   ...
end
```

That is, a function is created and stored in a table field (one should remember that `object.foo` is equivalent to `object["foo"]`); moreover, this function has a hidden parameter called *self*.

A method call can be written as

```
receiver:foo(a1,a2,...)
```

which is translated to

```
receiver.foo(receiver,a1,a2,...)
```

In other words, the receiver of the method is passed as its first argument, giving the expected meaning to the parameter *self*.

Part of the reflexive nature of Lua is provided by fallbacks. Fallbacks allow programmers to set their own functions to handle error conditions. Fallbacks are also used to provide hooks to handle other situations that are not strictly error conditions. This more general use of fallbacks is one of the cornerstones of LuaOrb implementation, the binding between CORBA and Lua. Although the semantics of fallbacks are quite similar to exception handling with resumption, the dynamic nature of Lua allows their use in many cases where a statically typed language would issue an error at compile time. As a typical example, one should consider the following statement:

```
obj:foo(a,b)
```

which is equivalent to:

```
obj["foo"](obj,a,b)
```

Usually, obj would be a table, and therefore `obj["foo"]` would be the field `"foo"`of this table. However, `obj` does not need to be a table; it can be a userdata, for instance. In this case, a user defined fallback will be called, and this fallback will be responsible for providing a sensible value to the expression `obj["foo"]`. If this value is a function, this function will then be called with the parameters `obj`, `a` and `b`. Otherwise another fallback will be called, to handle the "invocation" of this non-function value.

6. LUAORB

Since CORBA objects are to be accessed like any other Lua object, the generation of stubs is neither necessary nor interesting. Instead, CORBA objects should be accessible from Lua with no need of previous declarations, and with dynamic typing. In order to accomplish this goal, the binding is built upon the dynamic interface.

To explain the way Lua uses CORBA objects, the following example of an IDL interface is used:

```
struct Cuboid
{
  string cuboid_id;
  string material;
  string partnumber;
  double size;
}
```

```
interface CuboidWrapper
{
  boolean AddCuboid(in Cuboid
    cub_info);
  void ChangeSize(in double new_size);
  double GetSize();
  void AddRelated(in srting
    related_cub_id);
};
```

After a proxy of a distributed object is created by the function *createproxy*, the services of the related object can be requested. For instance, the method `AddCuboid` of the `CuboidWrapper` interface can be called as follows:

```
new_cub = {cub_id= "FrontLeftCub",
           material = "steel",
           partnumber = "A201",
           size = "2.54"}

new_cuboid =
createproxy("CuboidWrapper")

result =
new_cuboid:AddCuboid(new_cuboid)
```

The Lua table `new_cuboid` is dynamically mapped to the IDL structure `Cuboid` , when `AddCuboid` is called. Since `new_cuboid` is not a table, the expression

```
new_cuboid["AddCuboid"]
```

triggers a fallback. The fallback function, defined by LuaOrb, simply returns the string `"AddCuboid"`, which will be used by Lua as the "function" to be called. When Lua tries to call this string, another fallback is called, as explained in the previous section of this paper. This second fallback receives as parameters the string `"AddCuboid"` (the method name), the userdata `new_cuboid` (the proxy reference), and the table `new_cub`. With this information, the LuaOrb fallback asks the Interface Repository (where the IDL definitions for all objects are stored) whether the IDL interface `CuboidWrapper` has a method named `AddCuboid`. If this is the case, the description of `AddCuboid` (parameters, context information, result and exceptions) is retrieved from the Interface Repository.

With the Lua parameters and the method description at hand, the parameter list for the dynamic call is built. The Lua parameters are adjusted to the types of the IDL description. After that, the method `AddCuboid` is invoked. After the execution of the method, LuaOrb checks whether some exception was raised. If there are no exceptions, the method's returned value and result parameters are converted to Lua values and returned to the client application. Otherwise the exception is raised.

7. THE CORBA MODEL FOR VIRTUAL PROTOTYPING

Figure 6 illustrates the proposed CORBA model in which an application module is simultaneously client and object implementation. In this model, The Abstract Data Object (ADO) is a C++ class containing the object's geometry data (ACIS). The class *wrapper* is the ORB skeleton, i.e. it is the server, with methods that can be called via CORBA. These methods access the ADO and also communicate with the Lua code to update the user interface. *Proxy* is a LuaOrb object through

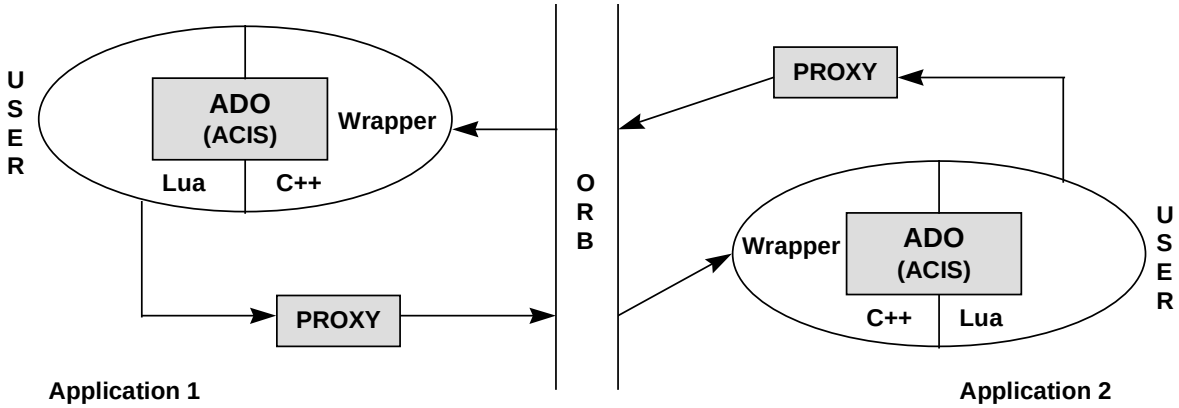

Figure 6. CORBA model for Virtual Prototyping

which Lua accesses the methods of a remote object via CORBA.

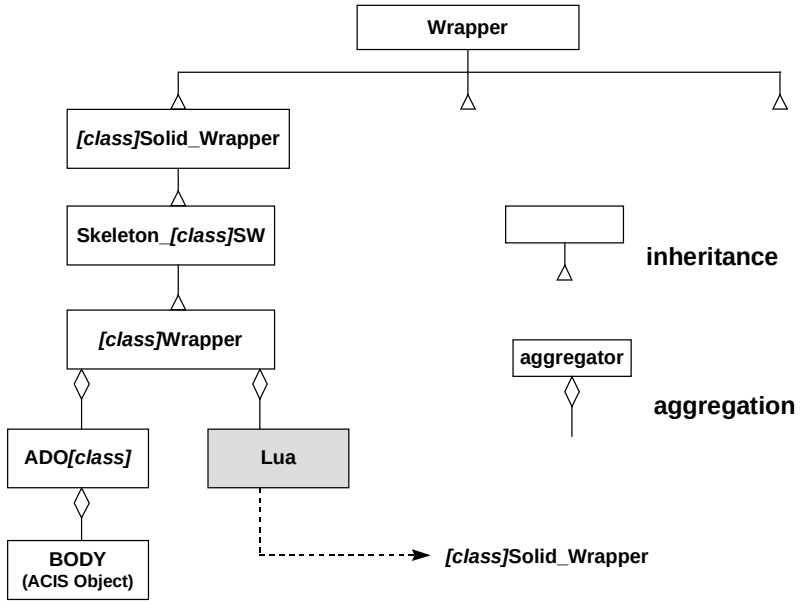

Figure 7. Class diagram of the proposed CORBA model

In the proposed model, each ACIS solid class is wrapped with a specific solid wrapper, as shown in the class diagram of Figure 7. The diamonds in this class diagram denote *aggregation*, in the sense that a class owns or is responsible for another one. In this case, one may say that a class "has" or "is part-of" another class. In the diagram of Figure 7, *[class]Solid_Wrapper* is a stub containing only declarations about the basic functions required by CORBA and the interface of the methods for a specific class (e.g. `BoxSolid_Wrapper` is the stub for the class Box.). The *Skeleton_[class]SW* is the server-side equivalent of the client stubs, created by the IDL compiler to bridge between ORB and object-implementation in *[class]Wrapper*. There is a skeleton for each solid type bound to the ORB. For instance, in the case of the type Box, `Skeleton_BoxSW` bridges between ORB and the services wrapped by `BoxWrapper`.

[class]Wrapper (e.g. `BoxWrapper`) is composed of C++ classes and has two functions: to wrap *ADO[class]* and to link to the Lua table. The Lua table contains pointers to the remote object (*link-to* relations) and the methods of the Extended Constraint Graph (e.g. `t = r/2`). It is through the Lua table that services are requested from remote objects. The *BODY* is the highest topological class in ACIS. The pointer to *BODY* allows a quick traversal of solid data.

The most important characteristic of the proposed model is its capacity of changing properties at run time using Lua. In order to illustrate this characteristic, an example was implemented according to the CORBA model proposed above. It is able to create simple solid models made of ACIS box or cylinder solids. As a start point, the ORB must be started on the network, then a first instance of the example application shall be started and a model must be created as a free solid, (e.g. without any constraint relations). After its creation, this free solid is able to act as a CORBA server object, thus its ready to respond to another object's request through ORB. Then, a second instance of the application can be started. Its solid model can be created either as a free or as a related solid. In this example, a related solid means an object that has some constraint relation with some remote object. When the second option is chosen, a two way client/server relation is established between the local and the remote objects. Conformity with the CORBA model proposed, this relation enables each object to behave as a client and as a server. A text edit box inside the application's main window can be used as a Lua console window, therefore Lua expressions are interpreted at run time. Due to the fact that the CORBA client calls are made from a Lua object through LuaORB, it works as a tool for changing properties and even relations at run time.

The following example illustrate this feature: two instances of the application are running simultaneously. A box object scales down the radius of a remote cylinder by 2/3 (Figure 8). In Fig.8, box1 executes through the Lua console the following operation that requests two services:

```
box.related_solid:ChangeSize(2*(
        box.related_solid:GetSize())/3)
```

which is equivalent to:

```
cyl1:ChangeSize(2*(cyl1:GetSize())/3).
```

GetSize gets the actual value of the cylinder's radius and ChangeSize changes the value of the radius. These methods are implemented in ADOBox and BoxWrapper.

The example presented above runs under the Windows NT 4.0 platform and was developed with Microsoft's Visual C++ 4.0 using Lua 3.0 and ACIS 3D Toolkit 2.1 libraries. The CORBA implementation is Visigenic's VisiBroker for C++ 2.0 (Visigenic, 1997). The appendixes A, B and C present code fragments which illustrate details of the implementation.

8. CONCLUSIONS

Virtual Prototyping becomes much more interesting when 3D solids are dynamic distributed objects on heterogeneous platforms. However, the dynamic nature of the design process requires a more formal framework for reactivity and pro-activeness embracing 3D objects. This paper proposes the concept of Extended Constraint Graphs with bidirectional reactivity and one-way actions to represent reactivity and pro-activeness in virtual prototypes. Also, this paper presents an original approach to deal with the dynamic nature of virtual mock-ups through the use of the interpreted language Lua with a CORBA binding called LuaOrb. The CORBA model for virtual prototyping proposed in this paper is illustrated with a simple example where a CORBA object is modified at run time.

The term "Extended Constraint Graph" is proposed in this paper because it embraces the well-known Constraint Graphs found in the area of constraint-based, interactive applications (Zanden, 1996). However, the objectives of the

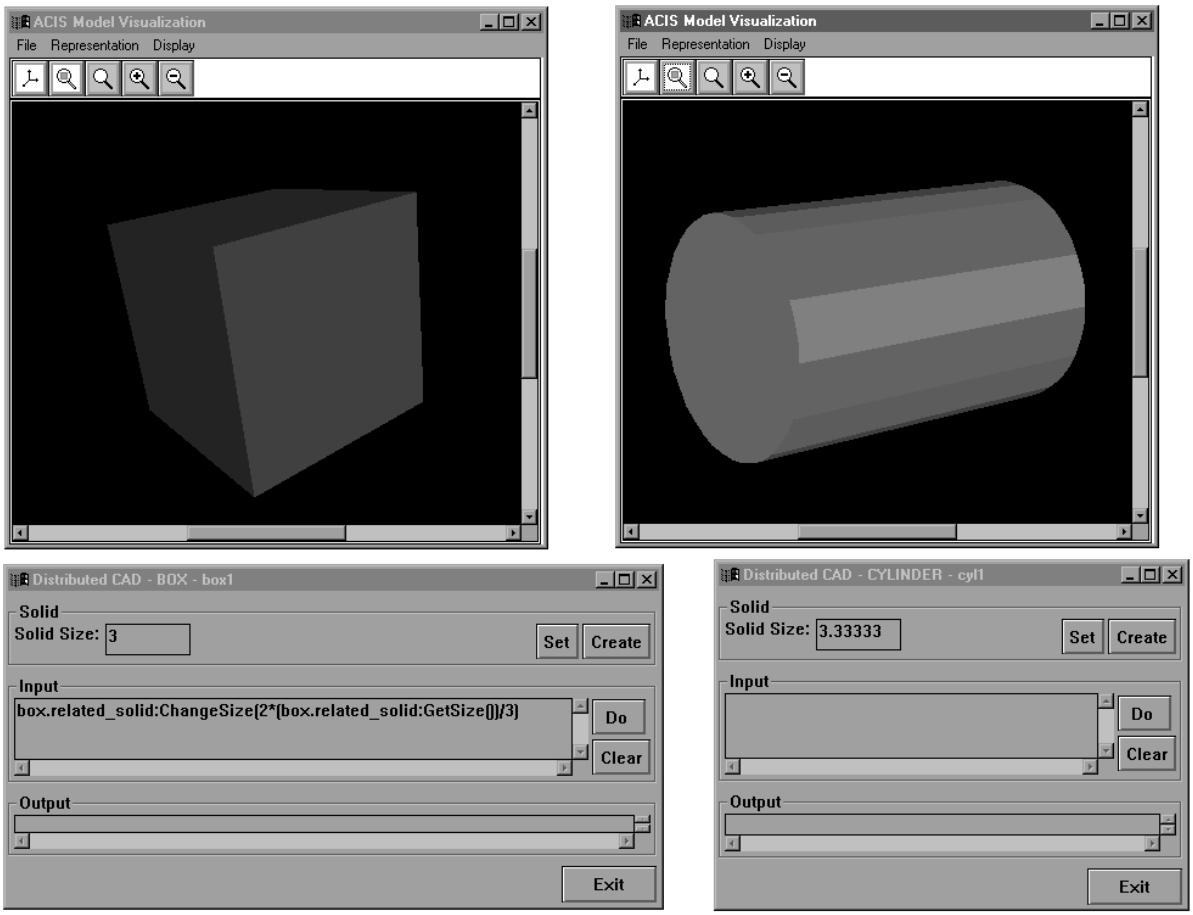

Figure 8. Modifying solids distributed on the network

present work are different from those found in the area of incremental constraint satisfaction. The ECG presented in section 2 was not fully implemented on a heterogeneous network, although the example of section 7 is a major step towards this full implementation. Nevertheless, several tests were made with the algorithm for bidirectional reactivity in order to validate the proposal.

Further work is needed to investigate the complexity of the algorithm for bidirectional reactivity. Another important topic actually being investigated by the authors is the development of an extension of LuaOrb to cope with transaction processing (Cobb, 1995). Also it is necessary to evaluate the power of the proposed mechanism on networks with heavy traffic.

ACKNOWLEDGMENTS

The authors would like to acknowledge CNPq for the financial support. Thanks are also due to Nick Lehtola for his initial insight into bidirectional reactivity. The authors would also like to thank João Bento and Sérgio Scheer for the valuable discussions and contributions. The authors also thank Embraer, the Brazilian aeronautics company, for the opportunity of testing these new ideas.

APPENDIX A

IDL file used in the example (*cube*Solid_Wrapper of Figure 7):

```
interface cubeSolid_Wrapper
{
  oneway void ChangeSize(in double new_size);
  double GetSize();
  oneway void AddRelated(in string new_solid);
  oneway void CChangeObjectValue(in double
new_value);
};
```

APPENDIX B

C++ header file for *cube*Wrapper and ADO*cube* presented in Figure 7.

```
//-----------------------------------------------------------------
// purpose---
//   define cubeWrapper class for interfacing an
//   ACIS cuboid body with CORBA enviroment.
//-----------------------------------------------------------------


#include "lua.h"
#include "wrapper_s.h"
#include "kernel/kerndata/top/body.hxx"

//-----------------------------------------------------------------
// Class ADO_cube:
// purpose: to set up ACIS data structure for the solid
// cuboid (a BODY structure). The cuboid is built in
// the function MakeBox.
//-----------------------------------------------------------------
class ADO_cube
{
 public:
   ADO_cube(double size);
   virtual void   ChangeSize (double new_size);
   virtual double GetSize ();

 private:
          BODY* m_pBox;
          double m_Size;
    void MakeBox(double size);
    void init();
};

//-----------------------------------------------------------------
//  The class cubeWrapper inherits Skeleton_cubeSW
// (i.e. _sk_cubeSolidWrapper) that is the CORBA
// server object generated by compiling the IDL code
// presented in APPENDIX A. This class has two
// important members: ado_cube and luaTable.
// LuaTable maintains a reference to the proxy
// written in Lua
//-----------------------------------------------------------------
class cubeWrapper : public _sk_SolidWrapper
{
 public:
   // Constructor and Destructor
   cubeWrapper(ADO_cube* cube, int luatbl, const
                  char *name=NULL);
   virtual ~cubeWrapper();

   virtual void   ChangeSize (double new_size);
   virtual double GetSize    ();
   virtual void   AddRelated (const char* new_solid);
   virtual void CChangeObjectValue(double
                                    new_value);

 private:
   int luaTable;
   ADO_cube* ado_cube;
```

```
};
```

APPENDIX C

Contents of the class ATT (in Lua code):

```
function ATT : new(label)

 local att = { }

 att.parent     = self
 att.Label      = label  -- Identification
 att.NumLin     = 0      -- No. of links to others obj
 att.LinkTo     = { }    -- adjacent nodes in the
                         -- graph
 att.attrib     = ""     -- name of the attribute
 att.this       = nil    -- reference to an ADO_
object (which contains an ACIS solid)
 att.wrapper    = nil   -- reference to a
              -- Solid_Wrapper object (CORBA server)
 att.update_cb = nil   -- function for interface
updating

 return att
end
```

Example of the service GetSize in Lua code.

This is a local function. The function called from a remote object via CORBA is written in C++ as presented after the Lua function.

```
-- ------ Local Lua Function - GetSize -------------
function ATT : GetSize()
  print(self.Label.."  ATT : GetSize")
  return _get_size(self.this)
end

// ---------------------- C++ function ---------------------
// Purpose---
//    Return size of the cube to the remote object
//     that requested the service.
//-------------------------------------------------------------
double bcubeWrapper::GetSize()
{
  return ado_cube->GetSize();
}
```

The C++ function above uses the following function in order to access the geometry data stored in the class ADO:

```
//-------------------------------------------------------------------------
// Purpose---
//    Return size of the cube.
//-------------------------------------------------------------------------
double ADO_cube::GetSize()
{
  return m_Size;
}
```

REFERENCES

Almasi, G., Suvaiala, A., Goina C., Cascaval, C. and Jagannathan, V., (1997), "TclDii: A TCL interface to the Orbix Dynamic Invocation Interface"; *http://www.cerc.wvu.edu/dice/iss/TclDii/TclDii.html*.

Bento,J. And Feijó,B., (1997), "An agent-based paradigm for building intelligent CAD systems", *Artificial Intelligence in Engineering*, 11, pp 231-244.

Cerqueira, R, Rodriguez, N. and Ierusalimschy, R., (1997), "Binding an Interpreted Language to CORBA", *II Simpósio Brasileiro de Linguagens de Programação*, pp. 23-36.

Cobb, E.E., (1995), "Objects and transactions: together at last", *Object Magazine*, Jan 1995, pp. 59-63.

Feijó, B.; Lehtola,N.; Bento,J. and Scheer,S., (1996), "Reactive design agents in solid modelling", in *Artificial Intelligence in Design'96* ed. by J. S., Gero and F. Sudweeks, Kluwer Academic Publ., pp. 61-75.

Golberg, A. and Robson, D., (1983), *Smalltalk-80: The Language and its Implementation*, Addison-Wesley.

Ierusalimschy, R., Figueiredo, L. H., and Celes, W. F., (1996) "Lua - an Extensible Extension Language", *Software: Practice & Experience* 26(6): pp. 635-652.

Inference Corporation, (1985), *ART Programming Manual*, Los Angeles.

Object Management Group, Inc., (1995), *The Common Object Request Broker Architecture and Specification; Revision 2.0*, Framingham, MA, USA.

Ousterhout, J. K., (1997), "Scripting: higher level programming for the 21st century", *http://www.sunlabs.com/people/john.ousterhout/scripting.html*.

Porter, S., (1995), "What's up with ACIS ?", *Computer Graphics World*, v.18, n.9, sep 1995, pp. 30-36.

Regli, W. C., (1997), "Intranet-enabled Computer-Aided Design", *IEEE Internet Computing*, v. 1, No. 1, Jan-Feb 1997, pp. 39-50.

[Also in http://www.computer.org/ internet/ backissu.htm].

Siegel, J., (1996), *CORBA Fundamentals and Programming*, John Wiley & Sons.

Spatial Technology., (1996), *ACIS 3D Toolkit Version 2.1 (manuals)*, Spatial Technology Inc., USA.

Stefik, M., Bobrow, D. and Khan, K., (1986), "Integrating access-oriented programming in a multi-paradigm environment", *IEEE Software*, IEEE Inc, New Jersey.

Visigenic, (1997), *VisiBroker for C++ (manuals)*, Visigenic Software, Inc., *http://www.visigenic.com/techpubs/*.

Widom, J. And Ceri, S., (1995), *Active Database Systems: Triggers and Rules for Advanced Database Processing*, Morgan Kaufmann, San Francisco, California, USA.

Zanden, B. V., (1996), "An incremental algorithm for satisfying hierarchies of multiway dataflow constraints", *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 1, pp. 30-72.