DISTRIBUTED AGENTS SUPPORTING EVENT-DRIVEN DESIGN PROCESSES

BRUNO FEIJÓ, PAULO C. RODACKI GOMES
Intelligent CAD Lab., Dept. of Comp., PUC-Rio, Rio de Janeiro, Brazil

JOÃO BENTO
Department of Civil Eng., Instituto Superior Técnico, Lisboa, Portugal

SÉRGIO SCHEER
Center for Studies in Civil Eng. (CESEC), UFPR, Curitiba, PR, Brazil

AND

RENATO CERQUEIRA
Department. of Computing, PUC-Rio, Rio de Janeiro, RJ, Brazil

Abstract. In event-driven design processes, a digital mock-up is made of entities which are more than 3D, passive, virtual objects; they are part of a society of agents whose behavior emerges from the interaction between these agents. This paper proposes the concept of Extended Constraint Graphs with bi-directional reactivity and one-way actions to represent reactivity and pro-activeness in a society of distributed design agents. Also, while the majority of CORBA applications in CAD systems uses static invocations, this paper presents an original approach to deal with the dynamic nature of event-driven design processes through the use of the interpreted language Lua with a CORBA binding called LuaOrb. Moreover, the use of ACIS as a Geometry Bus is presented in the context of the CORBA dynamic invocation interface.

1. Introduction

As contended by many authors, intelligent CAD systems do not yet populate designers' workbenches as one would expect to have happened long ago.

There are a number of known reasons that partially explain the situation: the too complex nature and ill-structure of the design process itself, the unavailability of appropriate software apparatus, the use of explicitly symbolic and too formal computational approaches requiring a great deal of anticipation and the neglecting of fundamentally important cognitive needs of designers, are just a few of the most popular ones.

Indeed, the vast majority of design systems proposed or developed for domains involving strong geometrical concerns, such as architecture, structural or mechanical engineering, have been relying on computational frameworks that can broadly be classified as, essentially, *rule-based*. Such fact may not only be attributed to existing design models, which appeal predominantly to a problem-solving interpretation of design that subsequently leads to a rule-driven implementation, but also to reasonably clear inadequacy of the representational tools available for describing relevant design processes.

The authors contend that less formal approaches to the conception and development of design systems – resorting more to a supportive role of the system and less on a design automation one – is a fundamental requirement for higher levels of acceptance and, therefore, success of ICAD systems. Such weakening of formalization may well be attained by subscribing to a strategy of producing design systems of a more "*event-driven*" nature rather than a "*rule-driven*" one.

Such a strategy would promote conformance with well known and widely identified cognitive needs of designers generally neglected by existing systems, such as the need to work at different levels of abstraction, or the necessity of being able to decompose solution, among many others (Bento, Feijó and Dowling, 1989), as much as it would cope with the enabling of freedom of sequence in the exploration of the design space (Boulanger, 1997). Furthermore, design systems that are predominantly event-driven would ease forms of human/machine interaction in which perception and cognition could play a decisive role and greatly enhance the degree of conformance with user requirements. An obvious illustration of such type of user interface is one that, for example, would enable the actual manipulation of objects being designed (like solids are manipulated in solid modelling environments); another one would consist on the adequate hosting of a constraint posting (or constraint relaxation) mechanism; a final example of utmost importance is one in which both the manipulation of objects and the posting of constraints could occur concurrently in a network of workstations, as in the case of the present work.

What the authors suggest for framing the implementation of better design systems is that they should be present to assist events of design that may require or trigger support, i.e. it is the very occurrence of actions within the design environment that originates support from the system. Examples of such actions are 1) the geometrical (or functional) manipulation of objects, 2) the posting of new constraint or changing of existing ones or 3) an explicit request for advice or assistance, among others. Typical examples of reactions to these events are: 1) changes in other directly or indirectly related design objects, 2) the indication of design space limits (Smith et al., 1996) or the relaxing of one or several constraints in order to alleviate or removing inconsistencies and allow the process to proceed.

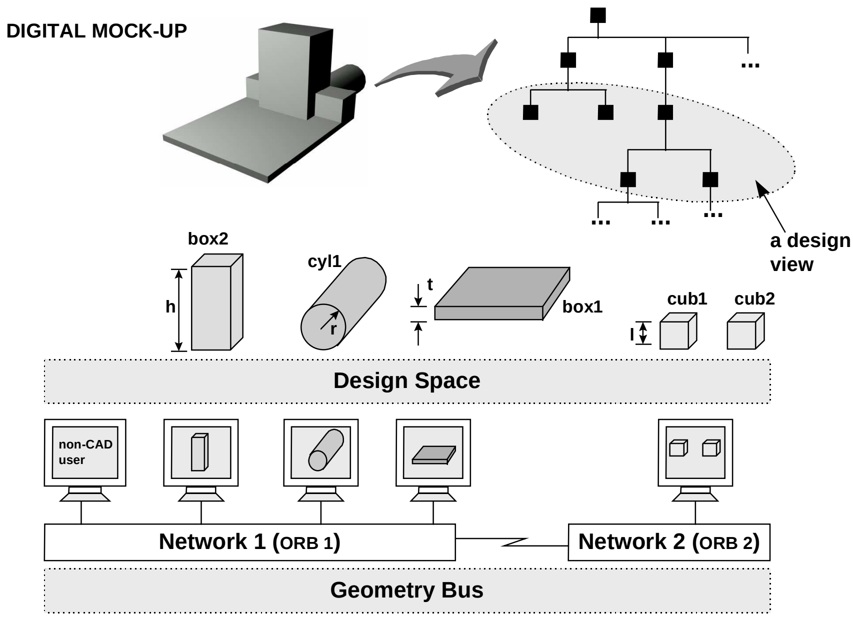

Figure 1 A Distributed Digital Mock-up

There are very few works on event-driven design systems in the literature and, consequently, design researchers miss this powerful paradigm in design. The present work resumes this theme starting from innovative applications of object-oriented distributed computing. Therefore, this paper presents a pragmatic model to support event-driven design processes starting from the following assumptions: (1) all product data are referenced to a digital mock-up; (2) the design environment is distributed; (3) the objects are dynamic, pro-active and reactive; (4) attributes and relations may not be fully known in advance and may also change at run time.

A digital mock-up is a virtual geometric prototype built by a conventional CAD system. Strictly speaking, a digital mock-up is one of the many views that one can extract from the design space. A ***design view*** is any set of objects and their links. The collection of all design views is called the ***design space*** (Figure 1). The objects in the design space should be reactive, pro-active and autonomous in order to meet the highly dynamic requirements of the processes embedded in this type of space. These objects go beyond the goals of the traditional object-oriented paradigm and are more adequately described as agents. In this paper they are called ***design agents***. In this context, solids are more than passive 3D virtual objects, that is: they are a society of agents whose behavior emerges from the interaction between these agents.

Design agents have several types of attributes, such as geometric attributes, design intent attributes, manufacturing attributes, cost attributes, part number references and documentation references. Furthermore, these agents are distributed over different networks.

As far as geometric attributes are concerned, design agents should be supported by a ***Geometry Bus*** which consists of the following items:

- a standard object-oriented geometric library;
- a component architecture; and
- a standard geometric data format.

It is clear from Figure 1 that the digital mock-up may also be distributed on the networks with different platforms and operating systems during the design phase. Therefore, the design environment supposes the use of object-oriented distributed computing.

In order to illustrate the dynamic nature of the agents in a distributed design environment, one may imagine a distributed digital mock-up made of four virtual 3D objects distributed in four different platforms but strongly committed to each other (box1, box2, cyl1 and cub1 in Figure 1). In this virtual world, any change in a 3D environment triggers reactions in other environments. Also, but most important, any user or agent may create a new agent or change relationships between agents at run time. For instance, the change of the radius of a cylinder *cyl1* in the digital mock-up may cause the agent *box1* to create a fifth agent called *cub2*.

Geometry buses and distributed objects are in the enabling technologies for event-driven design processes. ACIS (Spatial Technology, 1996) and CORBA (Object Management Group, 1995) have been proposed as a geometry bus and a distributed object architecture for integrated CAD systems respectively (Porter, 1995) (Regli, 1997). However, there is a great scarcity of references on ACIS-CORBA integration. Also, the majority of CORBA applications in CAD systems uses static invocations. In this particular, there are virtually no references on dynamic distributed objects in CAD systems. Moreover, there are very few references to reactivity and pro-activeness in solid modeling in the literature (Feijó et al, 1996). This paper covers those deficiencies and presents a model for distributed objects exhibiting reactivity and pro-activeness in event-driven design processes based on CORBA and ACIS.

2. Objects and Agents

There are some major differences between objects and agents. An agent is a software entity that exhibits autonomy, sociability, reactivity and pro-activity (Wooldridge and Jennings, 1995). Even when an agent is considered a physical one (e.g. a robot) the focus is on the software entities manipulating sensors and effectors. Shoham (1993) considers an agent as a specialization of the traditional object. This specialization is done by fixing the object's state in terms of mental components such as beliefs, intentions, choices, and commitments (i.e. an agent is an object with mental state). Moreover, while traditional objects communicate through unconstrained types of messages, agents distinguish between a certain number of communicative acts, such as informing, requesting, offering, accepting, rejecting, competing, and assisting one another (Shoham, 1993). This paper only explores the reactive and pro-active nature of agency within a distributed 3D environment. Furthermore, the adopted concept of reactive agent follows the principles of emergence and situatedness proposed by Brooks (1991).

The principle of emergence states that the intelligence of the reactive agent system emerges from the interaction of agents among themselves and with their environment. The principle of situatedness states that the intelligence of a reactive agent is situated in the world and not in any formal model of the world built in the agent. Therefore, a reactive agent uses its perception of the world rather than deductions based upon a symbolic representation of this world.

A reactive agent is an entity which may be represented by an independent program that knows everything about itself including its relationships with other agents. This is, in essence, the same definition of object in the traditional context of object-oriented programming. Therefore, the concepts of class, method and inheritance used in this paper are the same as in conventional object-oriented approaches.

The question of computational intelligence, not being within the scope of the present paper, is discussed elsewhere (Bento and Feijó, 1997). Nevertheless, it seems appropriate to invoke the metaphor of the "flock of birds" provided by behavioral animation authors (Reynolds, 1987) to illustrate how unpredictable choreographies of flocks of birds define a form of *emergent* collective behavior of simple objects executing very simple tasks defined by a small number of rules, such as "follow the leader", "keep close to others" or "avoid obstacles". The agency idea also follows the statement that complexity belongs to the external world. In this case, the focus should shift from the close-world assumption of any set of algorithmic rules to the open-world of interactive systems. Here, the appropriate metaphor is that illustrated by Simon (1982) with the example of the irregular behavior of an ant on a beach. In this example, the computing mechanism of the ant is supposed to be simple, but the ant's behavior reflects the complexity of the beach. This sort of argument have recently been discussed by computer scientists who propose that interaction is more powerful than algorithms (Wagner, 1997).

The paper argues that such emergence may occur in design processes supported by agent-based design systems. Furthermore reactivity amongst agents, or between the user(s) and the object(s) being designed, may be triggered by an *event*. As referred before, some of the basic types of events that might trigger actions and, possibly, direct support from the system, are the posting of a new constraint, the modification of existing ones or simply any change in any of the objects being designed across the network of workstations.

3. Extended Constraint Graphs

Activities in the design space have a highly dynamic nature, in the sense that the agents should be reactive and pro-active. Reactivity means that the agents

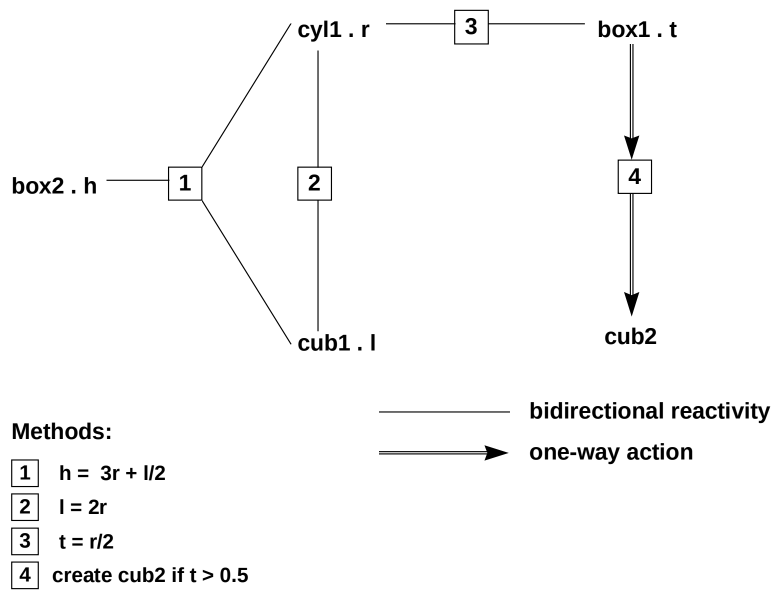

Figure 2 An Extended Constraint Graph

should be able to perceive the environment and react to changes in this environment. Pro-activeness means that agents act oriented by their goals and trigger actions automatically. Firing is inherent to the real-time nature of a society of dynamic distributed agents. This active behaviour is similar to those found in active database systems (Widom and Ceri, 1995) and is very useful in providing active support to the design process.

This paper proposes a special graph, called Extended Constraint Graph (ECG), to represent reactivity and pro-activeness in event-driven design processes, as an evolution of the authors' ideas found elsewhere (Feijó et al., 1996; Rodacki et al., 1997). A possible ECG for the virtual mock-up of Figure 1 is shown in Figure 2. A general 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:

agt[.attribute]

where *.attribute* is optional. Variables of the form *agt* denote the reference to an agent. Methods can represent a *bi-directional 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 agent. For instance, one may suppose that the actual 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 started by changing the height of box2.

Methods for bi-directional 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 agent should trigger other reactive links that this agent might have. Figure 3 illustrates this bi-directional 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.

One of the most important characteristics of the ECG approach is that the graph can be changed during the execution of the algorithm. Indeed, both variables and constraints may not only change dynamically, but also be created or removed along the process, since distribution (across a network of workstations) is a major assumption of the work. In this case, Extended Constraint Graphs cannot be handled by *offline* algorithms and, consequently, ECGs require *online* algorithms. An *online* algorithm must generate an output without knowledge of the entire input, because some relevant input data will arrive in the future and is not accessible at present (Albers, 1997). The case of ECGs gets worst because the constraints are generic, recursive and nonlinear relationships. The authors are not awared of any *online* algorithm in such an adverse condition.

Reactivity and pro-activeness 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 et al., 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. *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 agent *cub2* should be created if *t* > 0.5.

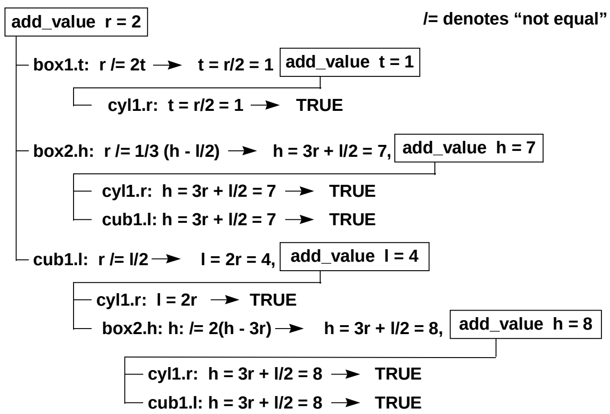

Figure 3 A Bi-directional Reactivity Process

Link-to relations make bi-directional reactivity possible. Every agent *i* linked to other agents has a list *link-to*$_i$ with members of the form:

$$< AGT_k.t_n,\ r_{ij}^{kn} >$$

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

$$AGT_i.t_j \text{ ---------- } AGT_k.t_n$$

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

```
function r_ijkn (AGTi, AGTk)
   get attribute Tj of agent AGTi
   get attribute Tn of agent AGTk
   get other attributes AT1, AT2, ... of other agents, if any
   if condition(Tj, Tn, AT1, AT2, ...) is not satisfied, then
          compute new value of Tn using inverse of condition
          return add_value(Tn to AGTk)
   end
   return TRUE
end of function
```

Before adding a new value to an attribute of an agent, the function `add_value` executes every function r_{ij}^{kn} found in the *link-to* list of the agent. This mechanism guarantees the full propagation of the changes in all directions.

In a design process, the Extended Constraint Graph of a digital mock-up 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.

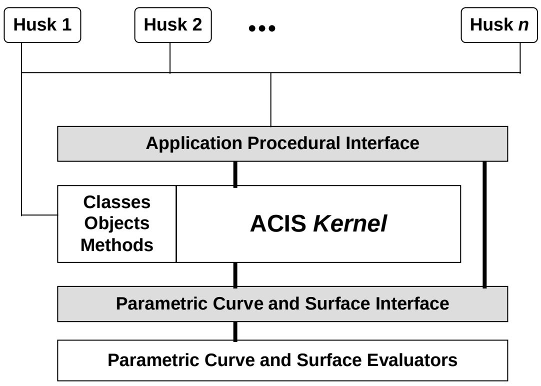

Figure 4. ACIS Architecture

4. ACIS as a Geometry Bus

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, according to the definition presented above, can be provided by ACIS architecture, ACIS components (called Husks) and ACIS SAT file format.

All products and applications built on top of 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 AutoCAD™, MicroStation™, CorelCAD™ 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, as defined above, 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.

5. 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 Smalltalk), 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 al., 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.

6. 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, 1997). Such configuration languages are also called extension languages.

Lua, an extension language developed in the Computing Department of PUC-Rio (Ierusalimschy et al, 1996), 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 a map function in Lua. A map receives a list `a=(a1,a2,...,an)`, represented as a table in Lua, 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))`.

```
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

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.

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.

7. 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 Cylinder
{
   string cylinder_id;
   string material;
   string partnumber;
   double r;
   double h;
}

interface CylinderWrapper
{
   boolean AddCylinder(in Cylinder cyl_info);
   double GetRadius();
   double GetHeight();
};
```

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 AddCylinder of the CylinderWrapper interface can be called as follows:

```
new_cyl = {cyl_id= “Shaft01”, material= “steel”,
            partnumber= “A201”, r= “2.54”, h= “5.08”}
new_cylinder = createproxy(“CylinderWrapper”)
result = new_cylinder:AddCylinder(new_cylinder)
```

The Lua table new_cylinder is dynamically mapped to the IDL structure Cylinder , when AddCylinder is called.

Since new_cylinder is not a table, the expression new_cylinder[“AddCylinder”] triggers a fallback. The fallback function, defined by LuaOrb, simply returns the string “AddCylinder”, 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 “AddCylinder” (the name of the method), the userdata new_cylinder (the proxy reference), and the table new_cyl. With this information, the LuaOrb fallback asks the Interface Repository (where the IDL definitions for all objects are stored) whether the IDL interface CylinderWrapper has a method named AddCylinder. If this is the case, the description of AddCylinder (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 AddCylinder 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 resulting parameters are converted into Lua values and returned to the client application. Otherwise the exception is raised.

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

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

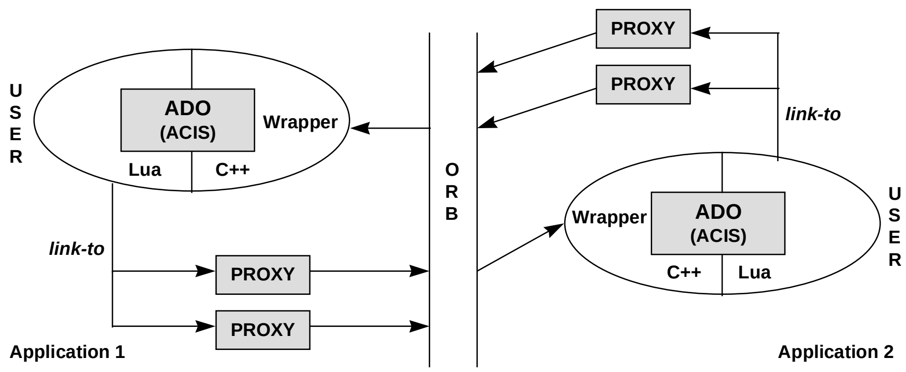

Figure 6. CORBA model for Virtual Prototyping

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 agent (*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 agents. The *BODY* is the highest topological class in ACIS. The pointer to *BODY* allows a quick traversal of solid data.

9. Examples

The most important characteristic of the proposed distributed agent model is its capacity of changing properties at run time as required by the design agents and the Extended Constraint Graph. In order to illustrate this characteristic, the example of Figure 2 is implemented according to the CORBA model proposed above. In this example, the system is able of creating a distributed digital mock-up made of simple solids like boxes and cylinders.

As a starting point, the ORB is started on the network, and then a first instance of the example is created with an agent in the form of a free solid (e.g. without any constraint relations). This free agent is then able to act as a CORBA server object, that is, it is ready to respond to another agent's request through the ORB. In the next step, any other agent can be created as a free or a related agent (i.e. an agent that has some constraint relation with some remote agent). When the second option is chosen, a two way client/server relation is established between the local agent and the remote one. In conformance with the proposed CORBA model, this relation enables the agent to behave as both a

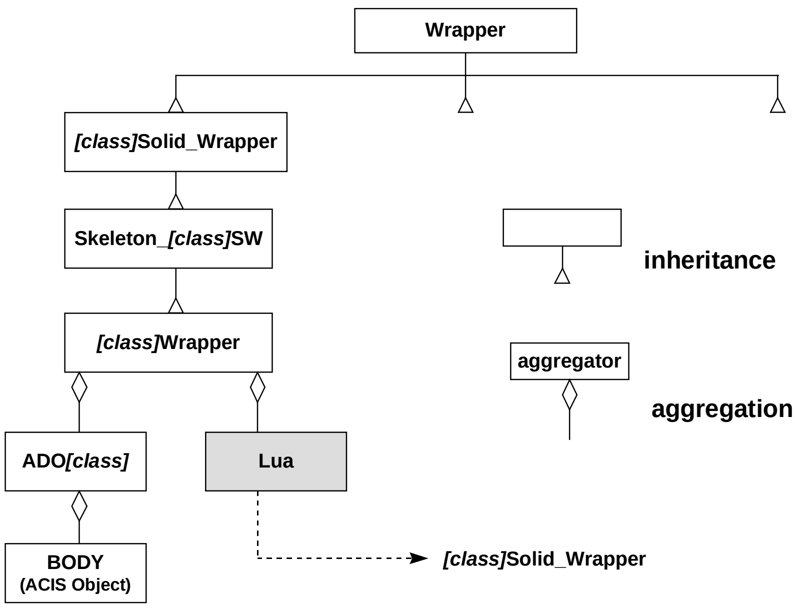

Figure 7. Class diagram of the proposed CORBA model

client and a server. A text editor box inside the application's main window can be used as a Lua console window, and Lua expressions are interpreted at run time. Lua expressions can be written by the user (or by an agent) in order to change properties and relations at run time.

The most impressive example is when the user or an agent changes the

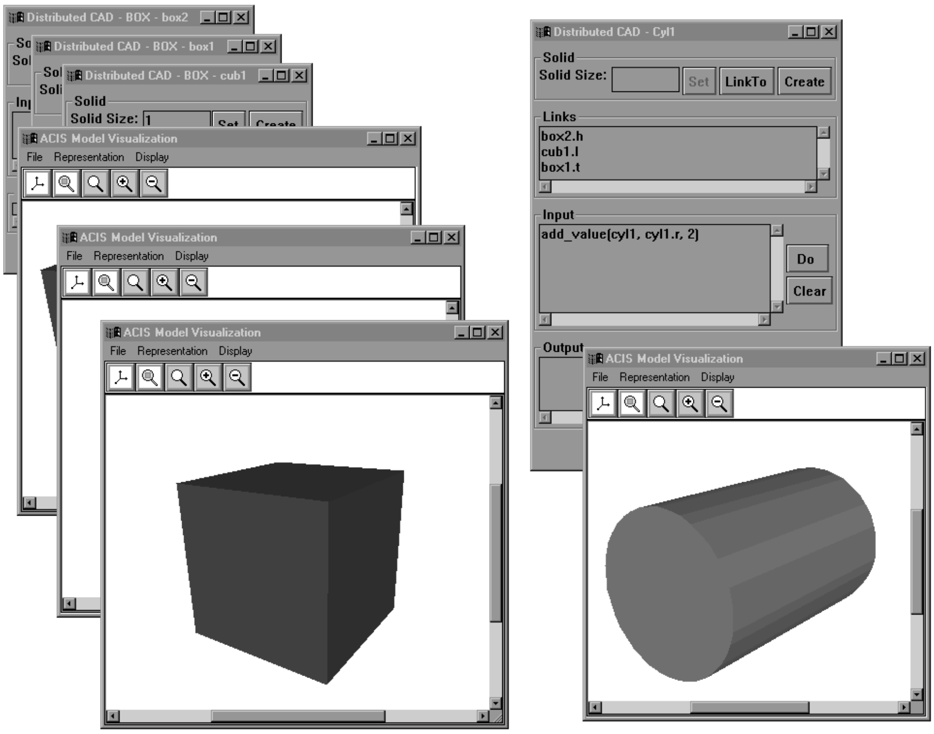

Figure 8. Solids as distributed agents

value of an attribute or an entire constraint equation at run time. In this case, the change takes place during the process of constraint satisfaction. Ordinary incremental constraint satisfaction processes cannot cope with this type of changing at run time. For instance, box1 might scale down the radius of a remote cylinder (i.e. the second solid in its *link-to* list) by 2/3 through the services `ChangeRadius` and `GetRadius`:

```
add_value(cyl1, cyl1.r, 2*(cyl1:GetRadius()/3))
```

where `GetRadius` is a method of cyl1 implemented in `ADOBox` and `BoxWrapper`. `GetRadius` gets the actual value of the cylinder's radius and `add_value` changes the value of the radius of cyl1.

Figure 8 presents another case where the user changes the radius of cyl1 writing the following expression (at run time) in the Lua console area of the window representing the solid cyl1:

```
add_value(cyl1,cyl1.r,2)
```

In this case, the status of the digital mock-up will be { r = 2, h = 8, l = 4, t = 1, cub2 created}. In this example, each agent is a solid and has an individual window for its visualization (using the Geometry Bus) - the two windows in the front displays cub1 and cyl1. Figure 8 also illustrates the testing environment developed by the authors.

The history of each attribute during the process depends on several factors, such as the order of the solids in the *link-to* lists. For instance, the evolution of the values of each attribute in the example above mentioned is:

$\{ r = 1 \rightarrow 2,\ h = 4 \rightarrow 7 \rightarrow 8,\ l = 2 \rightarrow 4,\ t = 0.5 \rightarrow 1 \}$

where the *link-to* list of cyl1 is:

{ box2.h, cub1.l, box1.t }

as noticed in the window of cyl1 in Figure 8.

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).

10. Conclusions and Further Work

The present work presents a model to support event-driven design processes based on pro-active, reactive and interactive distributed objects. This paper focuses on implementation issues in order to make clear the potentials, but also the drawbacks, of this new approach to CAD systems. The authors believe that this work reopens interesting discussions about event-driven design processes and the principles of emergence and situatedness, despite their complex unsolved problems such as unpredictable outcomes, chaotic behavior, oscillation and inconsistency. The authors do not claim that emergence and situatedness are sufficiently considered in the present work. These are principles underlying the work and they emerge only from the reactive and pro-active nature of the proposed model.

In virtual prototyping, a digital mock-up is made of entities which are more than 3D, passive, virtual objects; they are part of a society of agents whose behavior emerges from the interaction between these agents. Enabling, supporting, negotiating and managing such interaction as well as that between users and the virtual prototyping environment is, therefore, of utmost importance. Systems aiming at supporting such type of design environment are to be notably event-driven, in the sense that support is to be invoked as a consequence of changes in the environment, i.e. (attempted) changes of the objects being designed or on the constraints known to affect them; these are some of the events that trigger support and further reactions.

This paper proposes the concept of Extended Constraint Graphs with bi-directional reactivity and one-way actions to represent reactivity and pro-activeness in a society of distributed design agents. It presents an original approach to handle the dynamic nature of digital mock-ups through the use of the interpreted language Lua with a CORBA binding called LuaOrb and ACIS as the Geometry Bus. The authors undertake entire responsibility for the definition of Geometry Bus found in this paper, which are more formal than the one presented by ACIS's developers. The distributed agent model for virtual prototyping proposed in this paper is illustrated with simple examples where agents, as dynamic CORBA objects, are effectively 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). From the point of view of resolving networks of constraints, the approach proposed in this paper – drawing on works in the area of *online* algorithms (Albers, 1997) – is powerful, but naive. However, the goals and type of objects considered in the present paper go beyond those found in the area of incremental constraint satisfaction (Zanden, 1996) and constraint-based 3D manipulation for assembly (Fernando et al., 1995) Moreover, the authors are not aware of any constraint satisfaction algorithm coping with nonlinear constraints which change at run time and are distributed over a network of workstations.

Several examples of modification in the graph of Figure 2 (at run time) lead to nonrealistic values, such as very small or negative numbers. This means that the algorithm for bi-directional reactivity should be extended to work with variables which are confined in an interval. Some other examples lead to a crash situation. The problem in these latter cases is due to the lack of transaction control. A full control over the transactions requires an extension of Lua on the top of the CORBA service called OTS (Object Transaction Service) in order to construct reliable applications. For instance, applications should consider *rollbacks* (operations indicating that the current transaction has terminated abnormally and its effects should be discarded) and *isolation* (i.e. intermediate states are not visible to other concurrent transactions). The basic references for transaction processing are (Cobb, 1995), (Siegel, 1996, Chapter 14) and (Gray and Reuter, 1992).

Further work is needed to investigate the complexity of the algorithm for bi-directional reactivity and to explore competitive analysis for *online* algorithms in order to analyse the performance of this algorithm. Moreover, the question of loop should be further investigated. Another topic of the utmost importance for further work is the development of an extension of Lua to cope with transaction processing. Also it is necessary to evaluate the power of the proposed mechanism on networks with heavy traffic.

From the point of view of real applications in design, further work should be carried out to develop a programming environment to build more complex and general applications. Finally, cooperative design work should be investigated in order to propose a model for negotiation between agents (and agents/user) and for the establishment of early sets of constraints in the early phases of design problems.

11. Acknowledgements

The authors would like to acknowledge CNPq, Brazil, and JNICT, Portugal, for the financial support. Thanks are also due to Nick Lehtola for his initial insight into bi-directional reactivity and to Roberto Ierusalimschy for creating the opportunities with Lua. The authors would also like to thank Raphael C. Barbosa da Silva and José Carlos Ogliari Pinto from Embraer, the Brazilian aeronautics company, for the permanent incentive towards the research collaboration between PUC-Rio and the aeronautics industry.

12. References

Albers, S., (1997), “Competitive online algorithms”, Optima - Mathematical Programming Society Newsletter, n. 54, June 97, pp. 1-8.

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.P.; Feijó, B.; Dowling, P.J., 1989: “The Knowledge Based Design of Steel Portal Frames for Agricultural Buildings”, IABSE Report, 58, 281-290, IABSE

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

Boulanger, S., (1997), “Preliminary Bridge Design Navigation System for Novices”, Doctoral Thesis, EPFL, Lausanne.

Brooks,R.A., 1991. Intelligence without reason, A.I. Memo 1293, Artificial Intelligence Laboratory, MIT, Massachusetts.

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.

Fernando, T., Fa, M., Dew, P.M. and Munlin, M., (1995), “Constraint-based 3D manipulation techniques within virtual environments”, in: R.A. Earnshaw, J.A. Vince and H. Jones (editors), Virtual Reality Applications, Academic Press, 1995, pp.71-89.

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

Gray, J. and Reuter, A., (1992), Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publishers, New York.

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].

Reynolds, C., (1987), "Flocks, herds and schools: a distributed behavioral model", Computer Graphics, v.21, pp. 25-34.

Rodacki,P., Feijó,B., Cerqueira,R. and Ierusalimschy,R., (1997), "Reactivity and pro-activeness in virtual prototyping", mimeo, Dept. of Computing, PUC-Rio, Rio de Janeiro.

Shoham,Y., 1993. Agent-oriented programming, Artificial Intelligence, 60(1), pp. 51-92.

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

Simon, H., (1982), The Science of Artificial, 2nd ed., MIT Press, Cambridge, Mass.

Sleator, D.D. and Tarjan, R.E.: "Amotized efficiency of list update and paging rules", *Communications of the ACM*, 28, 202-208, 1895

Smith, I., Stalker, R., Lottaz, C.: "Creating Design objects from cases for interactive spatial composition", *Artificial Intelligence in Design '96*, ed.s John S. Gero and Fay Sudweeks, 97-116, Kluwer Academic Publishers, USA, 1996.

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.

Wagner,P., 1997: "Why interaction is more powerful than algorithms", *Communications of the ACM*, 40(5).

Wooldridge,M.J. and Jennings,N.R. Intelligent agents: theory and practice, Knowledge Engineering Review, Vol. 10, No. 2, 1995

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.