TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

ACTA TECHNICA NAPOCENSIS

Series: Applied Mathematics, Mechanics, and Engineering
Vol. 69, Issue I, March, 2026

DYNAMIC CAD GEOMETRY GENERATION VIA COM AUTOMATION USING JACOB-COM BRIDGE IMPLEMENTATION IN AUTOCAD

Tiberiu Alexandru ANTAL

Abstract: *The paper implements a Java-based automation system that interfaces with AutoCAD via the Jacob COM bridge to create and animate a 2D-2R robotic manipulator model. The research demonstrates real-time kinematic simulation of a two-link planar robot with visual feedback, implementing industrial automation principles in a Computer-Aided Design (CAD) environment.*
Key words: *animate, automation, CAD, COM, Jacob, Java.*

1. INTRODUCTION

Windows Automation refers to the set of technologies, tools, and methodologies used to programmatically control and interact with Windows operating systems, applications, and user interfaces without manual intervention. It enables computers to perform repetitive tasks, simulate user actions, and integrate disparate applications automatically. COM Automation is a Microsoft technology that enables applications to expose their functionality and be controlled programmatically by other applications or scripts. It's essentially a "remote control" protocol for Windows applications that allows them to communicate and share functionality across process boundaries. Before COM, AutoCAD [6] existed as powerful but isolated software. Engineers used AutoLISP for scripting inside AutoCAD, but communicating with the outside world as sending data to Excel, pulling specifications from databases, or integrating with manufacturing systems required file exports, manual copying, and processes tending to cause mistakes. The introduction of COM support in AutoCAD R14 (1997) was a strategic revolution. It transformed AutoCAD from a standalone drafting tool into a connected node in the enterprise software ecosystem. Suddenly, AutoCAD could have conversations with other applications. This wasn't just a technical upgrade; it was a business necessity driven by the increasing complexity of engineering workflows where CAD data needed to flow into analysis software, bill of materials systems, and manufacturing planning tools. At its core, a COM interface is a contract or a formal promise about how software components will communicate. It is a strictly defined set of rules that both the caller and the implementer agree to follow. AutoCAD's COM [7] interface presents itself as a logical hierarchy, mirroring how users think about a drawing. At the top or base of the hierarchy sits the Application object, this is the gateway to everything. From there, we navigate to Documents, which contain the actual drawings. Each Document contains collections: ModelSpace where the 3D design lives, PaperSpace for layouts, Layers for organization, Blocks for reusable components, and more. The object model is a conceptual structure that describes how Autodesk's designers thought about organizing AutoCAD's functionality. This object model is elegant because it's intuitive; programmers manipulate CAD elements using the same conceptual structure that designers use when working manually in the software. The object model is a conceptual design that could be implemented in any language. The COM

Received: 29.01.26; **Similarities**: 17.02.26: **Reviewed**:16.02. /28.02.26: **Accepted** 12.03.26.

interface is a binary standard that any language can call.

2. THE ROLE OF THE BRIDGE BETWEEN JAVA AND Windows COM

Java and Windows COM are fundamentally different technologies with completely different ways of thinking, different rules, different memory management, and different execution models. They share no common ground. They cannot interact to each other directly. Java runs in a carefully controlled, safe environment called the Java Virtual Machine and there:

- Memory manages itself automatically through garbage collection - objects are created, used, and then magically disappear when no longer needed;
- Code is platform-independent, running anywhere from Windows to Linux to macOS;
- Everything is object-oriented, with a single inheritance model
- The system is extremely restricted, isolated for security
- Developers never touch raw memory or pointers

COM exists in the raw, native Windows environment where:

- Every piece of memory must be manually managed for with reference counting;
- Components are tied to specific Windows versions and architectures;
- Multiple inheritance is common through interfaces;
- Direct memory access and pointer manipulation are everyday activities;
- Threading follows strict rules about how objects communicate.

Java's automatic memory management conflicts directly with COM's manual reference counting. Java's platform independence contradicts COM's Windows - specific nature. Java's security restrictions prevent the direct system access that COM requires. A translator is required between these technologies (worlds) so they could communicate. Jacob performs several layers of translation simultaneously:

- Memory Management Translation: In COM, every object must be reference-counted: when you obtain an object pointer, you call "AddRef" to increment its count; when you're done, you call "Release" to decrement it. If the count reaches zero, the object destroys itself. Java knows nothing of this. In Java, objects simply exist until the garbage collector decides they're no longer needed. There's no concept of "AddRef" or "Release." Jacob solves this by acting as a shadow accountant. When Java code gets a COM object through Jacob, Jacob secretly calls "AddRef" on your behalf. When the Java object is eventually garbage-collected, Jacob's cleanup code calls "Release." This happens invisibly, automatically, preventing what would otherwise be catastrophic memory leaks or premature object destruction.
- Data Type Translation: COM uses Windows-specific data types that don't exist in Java. A "BSTR" in COM is not the same as a Java String. A "VARIANT" can hold any type of data but needs special handling. A "SAFEARRAY" is a particular way of organizing arrays that Java doesn't understand. Jacob converts all these types back and forth. When you pass a Java string to a COM method, Jacob converts it to a BSTR. When COM returns a VARIANT containing a double, Jacob extracts the value and gives you a Java Double object.
- Threading Model Translation: COM has a complex threading model called "apartments." Some COM objects (like most UI applications including AutoCAD) must live in a Single-Threaded Apartment, meaning all calls to them must come from the same thread. Other objects can live in Multi-Threaded Apartments. Java threads know nothing about COM apartments. Jacob handles this by setting up the correct apartment model and marshaling calls between threads when necessary. That's why your code calls `ComThread.InitSTA()` - it's

telling Jacob to initialize the Single-Threaded Apartment that AutoCAD requires.

- Error Handling Translation: COM reports errors through numeric codes called HRESULTs. Java uses exception objects. When a COM method fails, Jacob catches the HRESULT, interprets what went wrong, and throws an appropriate Java exception with a meaningful message.

Jacob is more than just a library - it's an enabler of cross-platform, cross-technology solutions. It respects Java's philosophy of write-once-run-anywhere while acknowledging the reality that much of the business and engineering world runs on Windows with COM-based applications.

3. BASIC Java - AutoCAD INTERACTION USING Jacob

The **first step** is to call `ComThread.InitSTA()` which **prepares the communication channel** following Windows' specific requirements. The **second step** is to **find AutoCAD**. For this a dual connection strategy is used: try existing instance first, launch new one if needed. To connect to a running AutoCAD instance we write:

```
ActiveXComponent acad =
ActiveXComponent.connectToActiveInstance("
AutoCAD.Application");
```

To start a new instance we write:

```
ActiveXComponent acad = new
ActiveXComponent("AutoCAD.Application");
```

Once connected to AutoCAD the **third step** is to **go to where the drawing happens**:

- get AutoCAD handle;
- get the list of drawings or current;
- go to the ModelSpace, the 3D drawing area.

```
Dispatch app = acad.getObject();
Dispatch    doc    =    Dispatch.get(app,
"ActiveDocument").toDispatch();
Dispatch modelSpace = Dispatch.get(doc,
"ModelSpace").toDispatch();
```

The **fourth step** is about **creating the geom**etry. Java sends drawing commands through Jacob like "Draw a line from (0, 0, 0) to (10, 0, 0)".

```
Dispatch link1 = Dispatch.call(modelSpace,
"AddLine",              makePoint(0,0,0),
makePoint(10,0,0)).toDispatch();
```

Jacob translates these Java commands into AutoCAD's native language and delivers them. The **fifth step** is about **modifying in real-time the components** for the animation, for this:

- calculate new positions using math formulas;
- send updates through Jacob like "Move that line here";
- tells AutoCAD to refresh the display so the changes will be updated.

```
// Calculate new positions for link 1
// in the j1x, j1y, e1x and e1y variables
// using some math
// Update link1
Dispatch.put(link1,          "StartPoint",
makePoint(j1x, j1y, 0));
Dispatch.put(link1,            "EndPoint",
makePoint(e1x, e1y, 0));
//Regenerate the drawing
Dispatch.call(doc,      "Regen",      new
Variant(1));
```

Jacob [8], [9] ensures these updates happen smoothly, handling all the translation and communication behind the scenes. The concepts to understand the pieces of code are related to:

- `ActiveXComponent` is essentially a COM class factory wrapper. In COM terminology, a class factory is an object that creates instances of other COM objects. When we instantiate ActiveXComponent("AutoCAD.Application"), we're calling the CoCreateInstance() or GetActiveObject() Win32 API calls through JNI.
- `Dispatch` wraps an IDispatch pointer and implements the IDispatch::Invoke() calling mechanism through JNI. Every COM object in AutoCAD (Document, ModelSpace, Line, Circle) is represented as a Dispatch object in Java.

A typical (commented) code to draw two moving line in AutoCAD is:

```
import com.jacob.activeX.ActiveXComponent;
import com.jacob.com.*;

public class AcadManipulator {
 public static void main(String[] args) {
  ComThread.InitSTA();
  try {
   ActiveXComponent acad;
   try {
   // Attempt to attach to the ALREADY OPEN
AutoCAD instance
   acad                                    =
ActiveXComponent.connectToActiveInstance("
AutoCAD.Application");
   System.out.println("Connected to active
AutoCAD instance.");
   } catch (Exception e) {
   // If not open, start a new one
   System.out.println("AutoCAD not found.
Starting new instance...");
   acad                =                new
ActiveXComponent("AutoCAD.Application");
  }

  acad.setProperty("Visible",          new
Variant(true));

  Dispatch app = acad.getObject();
  Dispatch    doc    =    Dispatch.get(app,
"ActiveDocument").toDispatch();
  Dispatch modelSpace = Dispatch.get(doc,
"ModelSpace").toDispatch();

  System.out.println("Operating          on
drawing:     "     +     Dispatch.get(doc,
"Name").getString());

  // Manipulator Dimensions
  double L1 = 60.0;
  double L2 = 50.0;

  // Create initial links at (0,0)
  Dispatch             link1             =
Dispatch.call(modelSpace,       "AddLine",
makePoint(0,0,0),
makePoint(L1,0,0)).toDispatch();
  Dispatch             link2             =
Dispatch.call(modelSpace,       "AddLine",
makePoint(L1,0,0),
makePoint(L1+L2,0,0)).toDispatch();

  // Loop to simulate movement
  for (int i = 0; i <= 360; i += 3) {
   double  theta1  =  Math.toRadians(i);  //
Shoulder joint
   double theta2 = Math.toRadians(i * 2.0);
// Elbow joint

  // Forward Kinematics
  double j2x = L1 * Math.cos(theta1);
  double j2y = L1 * Math.sin(theta1);
  double eex = j2x + L2 * Math.cos(theta1
+ theta2);
   double eey = j2y + L2 * Math.sin(theta1
+ theta2);

  try {
   // Update Link 1
   Dispatch.put(link1,          "EndPoint",
makePoint(j2x, j2y, 0));

   // Update Link 2
   Dispatch.put(link2,        "StartPoint",
makePoint(j2x, j2y, 0));
   Dispatch.put(link2,          "EndPoint",
makePoint(eex, eey, 0));

   // REGEN: Use 1 (acActiveViewport) for
speed
   Dispatch.call(doc,     "Regen",     new
Variant(1));
  } catch (ComFailException e) {

//  If  AutoCAD  is  busy  (user  clicking
around), skip this frame
  System.out.println("AutoCAD      busy...
retrying.");
  Thread.sleep(100);
 }

   Thread.sleep(40);
  }

 } catch (Exception e) {
  e.printStackTrace();
 } finally {
  ComThread.Release();
 }
}

private static Variant makePoint(double x,
double y, double z) {
  SafeArray        sa        =        new
SafeArray(Variant.VariantDouble, 3);
  sa.setDouble(0, x);
  sa.setDouble(1, y);
  sa.setDouble(2, z);
  return new Variant(sa);
 }
}
```

In the above code:

- `Variant` - is a universal container for data. Since Java and AutoCAD use

different data types, `Variant` wraps numbers, text, points, etc., in a way both can understand;

- `SafeArray` - is the coordinate package for points. When we need to send (X, Y, Z) coordinates, `SafeArray` packages them in the correct order for AutoCAD.

4. MANIPULATOR CREATION and SIMULATION IN Java FOR AutoCAD INTERACTION USING Jacob

The implemented application is a 2R planar manipulator (see Figure 1) with two active revolute joints as described in [1] - [5] which have been enhanced as follows.

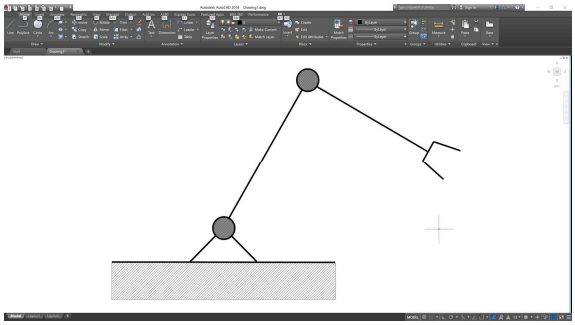

Fig. 1. - 2D manipulator representation in AutoCAD.

The code is enhanced to create an industrial-grade integration of robotic kinematics with CAD automation, moving beyond academic exercise to production-ready implementation.

4.1 Enhanced Visual Components

A **gripper block** creation with proper error handling is created. If the block already exists (common in re-runs) the code continues to run silently with the code:

```
String blockName = "GripBlock";
try {
 Dispatch gripBlock =
Dispatch.call(blocks, "Add",
makePoint(0,0,0), blockName).toDispatch();
 Dispatch.call(gripBlock, "AddLine",
makePoint(0, -4, 0), makePoint(10, -6, 0));
 Dispatch.call(gripBlock, "AddLine",
makePoint(0, 4, 0), makePoint(10, 6, 0));
 Dispatch.call(gripBlock, "AddLine",
makePoint(0, -4, 0), makePoint(0, 4, 0));
} catch (Exception e) { }
```

The **reference** or **base link** is created as the ground platform with scaled hatching. We create first a triangle then a close polyline, and then a new hatch pattern is created and added to the polyline. The **base link triangle** is created using:

```
double[] baseCoords = {-12, -12, 12, -12,
0, 0};
Dispatch basePoly =
Dispatch.call(modelSpace,
"AddLightWeightPolyline",
baseCoords).toDispatch();
Dispatch.put(basePoly, "Closed", new
Variant(true));
```

The **hatch** under the triangle is created using the code:

```
//create ground hatch contour as polyline
double[] rectCoords = {-40, -25, 40, -25,
40, -12, -40, -12};
Dispatch groundRect =
Dispatch.call(modelSpace,
"AddLightWeightPolyline",
rectCoords).toDispatch();
Dispatch.put(groundRect, "Closed", new
Variant(true));

//new hatch pattern
Dispatch hatch = Dispatch.call(modelSpace,
"AddHatch", new Variant(0), "ANSI31", new
Variant(true)).toDispatch();

//set pattern scale to 5.0 an create the
hatch
Dispatch.put(hatch, "PatternScale", new
Variant(5.0));
SafeArray saHatch = new
SafeArray(Variant.VariantDispatch, 1);
saHatch.setVariant(0, new
Variant(groundRect));
Dispatch.call(hatch, "AppendOuterLoop",
new Variant(saHatch));
Dispatch.call(hatch, "Evaluate");
```

The **revolute joint** representations are made using circles.

```
Dispatch.call(modelSpace, "AddCircle",
makePoint(0,0,0), jointRadius);
Dispatch joint2 =
Dispatch.call(modelSpace, "AddCircle",
makePoint(L1,0,0),
jointRadius).toDispatch();
```

5. CONCLUSIONS ON SYSTEM ARCHITECTURE

The implementation follows a four-layer modular architecture designed for maintainability and extensibility. The **COM Bridge Layer** utilizes the Jacob library to manage the complex translation between Java objects and COM interfaces, handling data organisation, reference counting, and thread apartment models (STA initialization). The **CAD Interface Layer** abstracts AutoCAD-specific operations through ActiveXComponent and Dispatch objects, providing a simplified API for document management, geometry creation, and property manipulation. **The Kinematic Engine** is a pure mathematical model that calculates joint and end-effector positions using trigonometric functions, completely independent of the visualization layer to ensure algorithm clarity and potential reuse. The **Visualization Layer** translates kinematic calculations into CAD geometry updates using efficient COM calls that modify existing objects rather than recreating them, enabling real-time performance.

6. REFERENCES

[1] ANTAL, Tiberiu Alexandru. *Using Java Affine Transformation in a Swing Based 2DOF Planar Robot Simulation.* Acta Technica Napocensis - Series: Applied Mathematics, Mechanics, And Engineering, v. 64, n. 1, 2021. ISSN 2393–2988.

[2] ANTAL, Tiberiu Alexandru. *Principles of Motion Simulation of a 2 DOF RR Planar Manipulator Using Java Swing*. Acta Technica Napocensis - Series: Applied Mathematics, Mechanics, and Engineering, [S.l.], v. 63, n. 1, 2020.

[3] ANTAL, Tiberiu Alexandru. *A Java Client-Server Model To Solve The Forward And The Inverse Robot Kinematics.* Acta Technica Napocensis - Series: Applied Mathematics, Mechanics, and Engineering, v. 62, n. 1, apr. 2019. ISSN 2393–2988.

[4] ANTAL, T .A., *Aspects concerning code reusabilty in planar mechanisms simulation, Acta Technica Napocensis*, Series: Applied Mathemathics and Mechanics, Nr. 48, Vol. I, 2005, p.45-50, ISSN 1221-5872.

[5] ANTAL, T .A., *An object based approach to planar mechanisms simulation, Acta Technica Napocensis*, Series: Applied Mathemathics and Mechanics, Nr. 48, Vol. I, 2005, p.37-44, ISSN 1221-5872.

[6] AutoCAD *ActiveX and VBA Developer's Guide (2023).* Autodesk.

[7] Lee Ambrosius, *AutoCAD Platform Customization: User Interface, AutoLISP, VBA, and Beyond*, Sybex, 2015, ISBN: 978-1-118-79890-4.

[8] Sheng Liang, *The Java Native Interface: Programmer's Guide and Specification*, Publisher: Addison-Wesley Professional, 1999, ISBN-10: 0201325772.

[9] Adam Nathan, *.NET and COM: The Complete Interoperability Guide*, SAMS, 2002, ISBN: 067232170X.

Un sistem de clasificare pentru traiectoriile mecanismului bielă manivelă

Lucrarea implementează un sistem automat de execuție de comenzi, bazat pe Java, care interacționează cu AutoCAD prin intermediul bibliotecii Jacob COM pentru a crea și anima un model 2D de manipulator robotic 2R. Cercetarea demonstrează simularea cinematică în timp real a unui robot planar cu două legături și feedback vizual, implementând principii de automatizare industrială într-un mediu de proiectare asistată de calculator (CAD).

ANTAL Tiberiu Alexandru, Professor, dr. eng., Technical University of Cluj-Napoca, Department of Mechanical System Engineering, antaljr@mail.utcluj.ro, 0264-401667, B-dul Muncii, Nr. 103-105, Cluj-Napoca, ROMANIA.