SAE Paper Number 2008-01-2262

Keys to Success with Knowledge-based Techniques

Dave Cooper
Product Development Head, Genworks International, Birmingham, MI USA

Michel van Tooren
Professor and Aerospace Department Chair, Delft University of Technology, Delft, The Netherlands

Wan Mazlina Wan Mohamed
PhD Researcher, Delft University of Technology, Delft, The Netherlands

ABSTRACT

Dating from the early 1980s, Knowledge-Based Engineering technology (KBE) has been used to capture and automate design and engineering, in particular in the automobile and aircraft industries.

A viable KBE system in the 21st century must provide users with a dynamic *modeling feedback loop* in an environment favorable to both exploration and experimentation, supplying various approaches for engineering a given set of artifacts. The fundamental properties of a KBE system must include automatic caching and dependency tracking for the scalable runtime performance of large models, minimal source code volume, and efficient and rapid tools for model development and debugging. And, not least, it must complement existing CAD systems.

A crucial aspect of a bonafide KBE system is its language-based core, embedded in a standardized, full-featured programming language, i.e., as a superset. The GDL (General-purpose Declarative Language) platform from Genworks achieves this by providing a domain-specific language (DSL) for KBE purposes, embedded in ANSI Common Lisp (often characterized as the "programmable programming language"). ANSI CL facilitates sophisticated code generation, written in a highly compact manner, which is then automatically expanded into the low-level CL, and finally machine code for execution.

This paper will consider the theoretical and practical issues associated with using such a language-centric approach for design and engineering automation. It also will address a number of the commonly heard objections from engineers and others to the use of a Lisp-based language, and respond to these largely unfounded objections.

The specific GDL platform by Genworks represents a contemporary and cost-effective KBE toolkit, affording all the significant benefits from the legacy, and very expensive, KBE systems, while incorporating a host of modern features, including: (1) portable web-based development and runtime environments, (2) compatibility of function with contemporary CAD and other data exchange formats, while (3) remaining free-standing from the proprietary CAD systems, with (4) robust underlying commercial components built into the package, to wit: (a) Allegro CL or Lispworks, and (b) SMLib surface/solid modeling. The paper will cover the specific features and benefits of Genworks GDL, as well as contrasting GDL to a number of alternative systems currently on the market.

Finally, this paper will explore a number of actual GDL applications, including a conceptual aircraft assembly, an aircraft wiring harness configurator, and an educational tool for modeling aerodynamic dragster models which provides a good example of KBE deployed using "web 2.0" AJAX techniques. The paper will also cover an AJAX-enabled browser-based graphical development environment for Genworks GDL called ta2.

INTRODUCTION

Knowledge-based Engineering (KBE) may be described as "engineering on the basis of electronic knowledge models."(1)

For approximately 30 years, language-based KBE has been applied to challenging design and engineering problems, mainly in very capital-intensive industries such as automotive, civil engineering, and, in particular, aerospace. A prototype successful use of KBE has been a long-running and highly productive application at the Boeing Company for generating the geometry of thousands of stringer clips fitted and shaped for precise locations in an aircraft fuselage.(2)

However, until recently the aerospace use of KBE has been generally concentrated in those highly funded high-end industry niches. The major industry objections to a more widespread adoption of language-based KBE, notwithstanding its benefits, have traditionally been:

1. Prohibitive cost of the legacy systems;
2. A common perception of unfamilarity or difficulty on the part of users in the use of Lisp as the modeling language;
3. A persistent belief that KBE could not be used in tandem with other widespread "mainstream" technologies already in place, notably the ubitiquous CAD systems encountered in every design shop.

The theme of this paper is to dispel these three misperceptions, and, to the contrary, demonstrate that none of them any longer pose legitimate objections to the widespread use of KBE technologies, an adjunct use that would undeniably serve to greatly enhance, and shorten, industry design capabilities.

A REVIEW OF THE STANDARD KBE FEATURES

INTRINSIC FEATURES COMMON TO BONAFIDE KBE SYSTEMS The following five essential "lowest common denominator" features are *intrinsic* in any generative KBE system:

Functional Coding Style: programs (i.e. models) return values, rather than modifying things in memory or in the model.

Declarative Coding Style: there is no "begin" or "end" to a KBE model – only a description of the items to be modeled.

Runtime Value Caching and Dependency Tracking: the system computes and memorizes those things which are required – and only those things which are required (no more, no less).

Dynamic Data Types: Slot values and object types do not have to be specified ahead of time. They are inferred automatically at runtime from the instantiated data. Their datatypes can also change at runtime. In fact, the entire structure and topology of a model tree can change, depending on the inputs.

Automatic Memory Management: When an object or piece of data is no longer accessible to the system, the runtime environment automatically reclaims its memory.

Refer to the authors' 2007 AIAA paper (3) for concrete examples of how these features manifest themselves in a present-day KBE system.

IMPERATIVE ADDITIONAL FEATURES FOR 21ST CENTURY KBE DEVELOPMENT The features outlined above are the rudimentary elements for a system to meet the textbook definition of KBE. However, for a system to deliver practical and marketable results, and integrate well with the contemporary engineering computing environment, it needs further capabilities:

Code-generating Macroexpansion: Allows for simple, compact surface-level code structure;

Full-featured Development Environment: Enables rapid model development and instant feedback on changes;

Long-term Viability and Openness of Platform: A concise, standardized and easily manipulated modeling language allows models to survive changes in computers, operating systems, and user interfaces;

True Code Compiling: High-level model code is translated into low-level executable machine code;

Tight Connection with NURBS Surface and Solids Geometry Kernel: Provides "best of both worlds" of programmatic and constructive approaches to modeling. The geometry kernel must support standard neutral CAD formats (e.g. IGES, STEP) in order to act as a gateway between the KBE system and various engineering analysis tools (for Finite Element analysis, Computational Fluid Dynamics, etc).

These additional features are all exhibited in a modern KBE system, as exemplified by the Genworks GDL tool.

EASE OF USE OF THE DOMAIN-SPECIFIC LANGUAGE

Contrary to the claim of a few critics, a well-designed KBE system does not have to be difficult to use. To the contrary, it offers the opportunity to bring an engineer's work "to life," increasing the ease and lowering the risk of exploring different design options for engineered systems. And, once an initial KBE application[1] is in place, it can thereafter be refined and improved indefinitely – the simple and uniform language-based representation keeps all the design intent consistently accessible, both by engineers and machines for processing indefinitely into the future.

The reputed difficulty with the use of a KBE language should not be a deterrent – to the contrary, it is the *ease* of use that should be the main compelling feature of a KBE system. That is, the main purpose of a KBE system is to provide an open-ended language to an engineer that sacrifices none of its power or flexibility, but which allows him

[1]We use the term "KBE System" to refer to a development framework, such as ICAD or Genworks GDL. "KBE Application" (while also a "system" in its own right) refers to a specific application written using a KBE System

to retain his identity as an engineer and not devolve into the role of a "hardcore" programmer or computer scientist.

For example, the aircraft assembly mentioned in this paper was put together within a few weeks by a Chilean student visiting the TU Delft in Holland, an individual who had had no previous KBE or programming exposure.

OVERCOMING RESISTANCE TO KEYBOARD "TYPING"

One of the very elementary objections we hear to the effective use of KBE is an aversion to typing on a keyboard. Engineers are often accustomed to the seeming point-and-click simplicity (and limitations) of traditional CAD systems and programming IDEs. Once that atavistic mindset is overcome, mastering the mechanics of a computer keyboard and editing system are analogous to an activity as basic as riding a bicycle – they need to be learned once and practiced for a while, but then bring a lifetime of benefits.

One could well argue that to claim overall "literacy" in the 21st century, one must be able not only to read and write but equally be able to operate a computer keyboard and text editing environment in an effective way. Gutenberg no doubt encountered a similar reluctance in response to the introduction of the printed word.

Once this reality of modern life is accepted, it makes sense to look for the most effective way to write and edit text. It happens that one of the common threads in KBE systems over the years has been to provide exactly this.

EMACS AND FIREFOX AS INTEGRATED DEVELOPMENT ENVIRONMENT

Gnu Emacs Of the multitude of choices available for writing and editing GDL code, the default editing environment is the venerable Gnu Emacs. Emacs is an extremely powerful and customizable editing environment which has deep hooks into programming languages, especially dynamic ones such as Lisp. Emacs admittedly does not afford much in the way of a colorful point-and-click mouse interface.

However, with a few dozen keychords (easily memorized) the most frequently used and most helpful interactive information can be accessed in a clean and friendly way. Furthermore, Emacs runs virtually the same on all modern systems (and several antique ones as well), making it highly ubiquitous (4).

Examples of the kind of information available through SLIME(5) and similar emacs-Lisp interface packages are:

- Color-coding of source code, customized for KBE language such as GDL
- Automatic lookup of documentation, function arguments, object input/output protocols, and other specifications, within a standard editing window
- Automatic completion of any recognized symbol (i.e. word) after typing just the first few characters
- Automatic indenting ("pretty printing") of source code
- Highlighting of matching parentheses and balanced parentheses detection

Firefox (or other standard Web browser) From Google's online office suite to full-blown creative tool suites, Web Browsers are fast becoming the default user interface and deployment mechanism for many types of software applications. With so-called "Web 2.0" techniques, with AJAX being a notable example, browser-based applications can provide a level of user interaction very nearly approximating desktop applications.

For KBE applications which, by definition, perform multiple tasks, such as synthesizing and presenting information from different sources, generating pictures of geometric models, and gathering responses from the user, a web-based deployment mechanism is the natural approach.

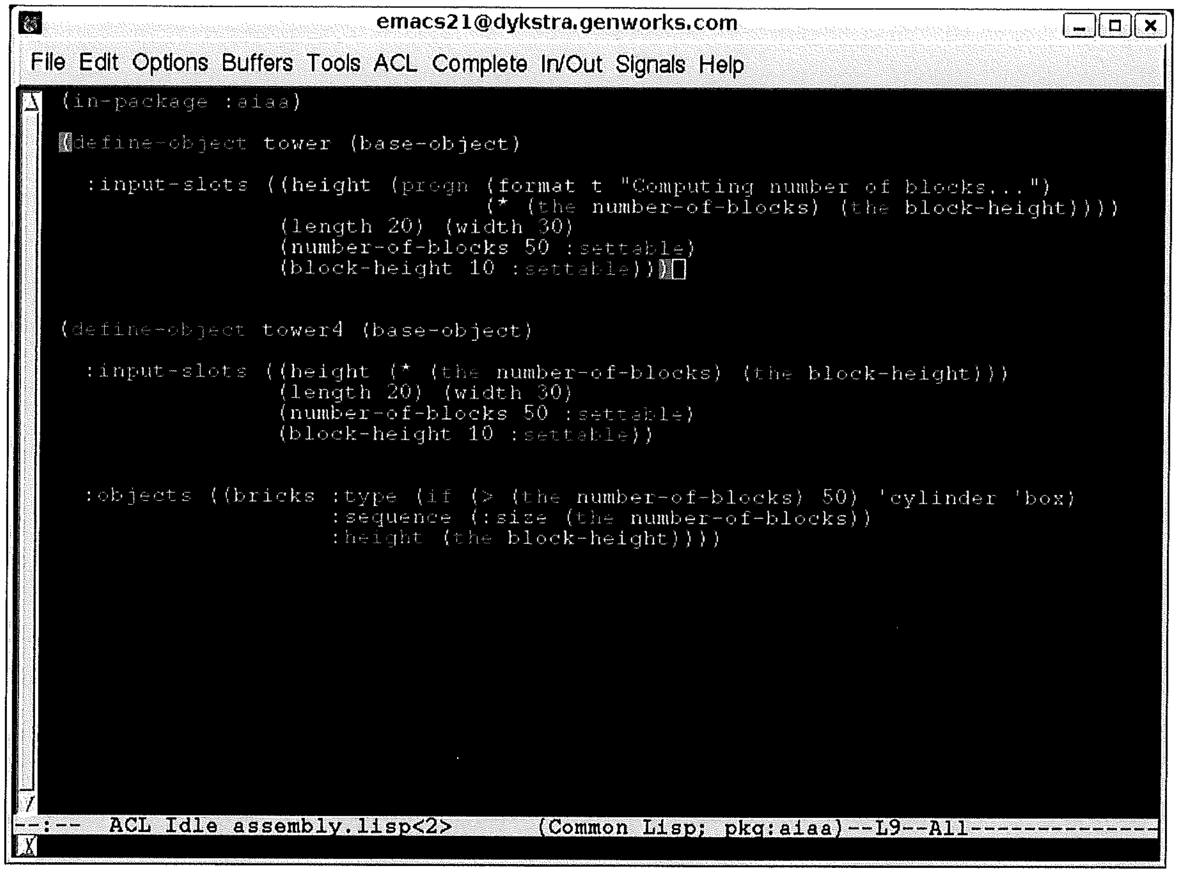
A Web-based deployment also insulates the application from depending on the particular operating system and computer type on the user's desktop.

Incremental, Closed-Loop Development Unlike most programming languages which demand constant re-running and debugging of the program, a KBE model has no explicit "begin" or "end." The user (e.g. engineer) can continually make updates to the model source code, and compile those updates into the system at the touch of a button, and instantly inspect the results.

This positions a KBE systems such as Genworks GDL, into a wholly different category when contrasted to traditional programming. With a KBE system, the objects to be modeled are simply "described" in a declarative fashion, using the domain-specific language. As a separate "Runtime" step, the system then *automatically* "runs" the code in the correct order to produce requested outputs.

PREFIX NOTATION (I.E. SYMBOLIC EXPRESSIONS, OR "S-EXPRESSIONS")

Lisp-based KBE languages typically use a form of Symbolic Expression ("S-Expressions") not only to capture the definition framework (e.g. `(define-object ...)`), but also consistently, throughout the language, to represent expressions down to the lowest-level function call or math formula. A Symbolic Expression ("S-exp" for short) consists of a list containing first, some kind of *operator*, followed by some number of *arguments*. The arguments themselves can likewise be S-expressions.


emacs21@dykstra.genworks.com
File Edit Options Buffers Tools ACL Complete In/Out Signals Help

```
(in-package :aiaa)

(define-object tower (base-object)

  :input-slots ((height (progn (format t "Computing number of blocks...")
                               (* (the number-of-blocks) (the block-height))))
                (length 20) (width 30)
                (number-of-blocks 50 :settable)
                (block-height 10 :settable)))


(define-object tower4 (base-object)

  :input-slots ((height (* (the number-of-blocks) (the block-height)))
                (length 20) (width 30)
                (number-of-blocks 50 :settable)
                (block-height 10 :settable))


  :objects ((bricks :type (if (> (the number-of-blocks) 50) 'cylinder 'box)
                    :sequence (:size (the number-of-blocks))
                    :height (the block-height))))
```

```
--:--  ACL Idle assembly.lisp<2>       (Common Lisp; pkg:aiaa)--L9--All-------------
```

Figure 1: Portable, Venerable Editing and Inspecting Environment

The language syntax is that basic. That is all the syntax you need to know.

An occasional point of contention heard from engineers (in particular), is the DSL's consistent use of prefix notation for math expressions.[2] So an expression like `2 + 2` in a "normal" language would be expressed as `(+ 2 2)` in a Lisp-based KBE language. While this infix notation takes a bit of getting used to, it makes sense to keep it in the language for several reasons:

Uniformity The syntax of the language is uniformly prefix, so there is no need to learn and memorize arbitrary rules of operator precedence[3]

Self-writing Code (i.e. macroexpansion) The uniformity of syntax is what allows the KBE system to write most of your code for you.

Decomposition of Complexity One of the winning features of a KBE system is its ability to decompose a problem into manageable parts. This includes math expressions. S-expressions can be moved around and decomposed like so many building blocks.

Furthermore, in actual practice, the use of simple arithmetic expressions typically make up only a small fraction of the overall application. Consequently, their influence on the "big picture" of ease-of-use is de minimus.

INTEGRATION WITH TECHNOLOGY INFRASTRUCTURE

KBE was originally restricted to the research laboratories of major corporations. Addressing its inability to interact with the rest of the "Enterprise" computing infrastructure (such as any existed in the early KBE era) was not a priority. However, given the advances of modern computer technology, a number of factors have converged which now enable KBE to blend seamlessly into that *mainstream* use. Specifically, we can point to the following critical developments:

1. ISVs have entered the arena with robust KBE development suites at *approachable cost levels.* The

[2]In fact, it is possible to support infix notation for mathematics and have the KBE system automatically transform this into its own prefix notation. However the cost of doing this is increased complexity and more "rules" to follow when writing code– in practice, to our knowledge, this has never proved to be of benefit to anyone.

[3]According to the renowned MIT author and professor Gerald J Sussman(6): "In functional notation mathematical expressions are unambiguous and self-contained." By "functional notation" he is referring to prefix notation, and his book *Structure and Interpretation of Classical Mechanics*(7) is devoted to describing physical mechanical systems using precisely this kind of notation. Sussman even goes so far as to claim that traditional (infix) notation is harmful to students' ability to grasp mathematical concepts (i.e. "it is not the concepts which confound them, it is the notation.")

Genworks offering, for example, incorporates the decades of legacy KBE experience blended with the mature, ANSI standard Common Lisp implementations (more on this later).

2. *Fast, inexpensive computers*, with ample memory resources to handle ever-larger model definitions and instantiations, have become commodity items.

3. The *World Wide Web* and associated standards such as HTML, XML, and X3D provide for cross-platform, royalty-free application deployment across an enterprise or across the globe.

CAD AND GEOMETRY INTEGRATION It is important to recognize that KBE is a wide-spectrum general-purpose programming and geometry modeling concept, as contrasted to being merely a sophisticated drafting or engineering analysis tool.

Historically there has been a conflict insofar as the role of KBE with respect to its interaction with traditional CAD systems. Whatever that past controversy regarding the respective roles of KBE vis-a-vis CAD, the reality of the engineering and design workplace today is that the workplace has become CAD dominated. Drumbeats such as "CAD the Master" were widely popular in the 1990's, and to a large extent, still are. But that is not the end of the story.

Realistically, in order to be successful in the modern marketplace, state-of-the-art KBE technology must *complement the existing CAD systems*, just as it must complement existing Database systems (e.g. Oracle, MySQL), delivery mechanisms (e.g. webservers and web browsers), and typesetting formats (e.g. PDF). For example, Genworks GDL accomplishes this transition through the use of a geometry kernel library which contains *full support for all the standard CAD formats*[4].

THE MARKETPLACE OF PACKAGED KBE TOOLS

Vendors today offer two principal categories of KBE systems:

1. The first category is comprised of what are in fact CAD systems masquerading as KBE, with the "KBE" tacked on as an additional whistle.

2. The second are the bonafide language-based, dedicated KBE systems, with geometry systems at their core.

CAD SYSTEMS WITH "BUILT-IN" KBE Prime examples of this first category are KnowledgeWare from Dassault Systems, and Knowledge Fusion from Unigraphics. These are CAD systems which purport to have KBE "built-in."

As an initial impression their hybrid makeup sounded promising, that is, to join the engineering knowledge and power of KBE with the ubiquitous CAD systems, which were already pervasive throughout industry. In short, at first glance, it had the aura of "this could be the best of both worlds."

But further and careful examination, when measured against KBE principles, reveals major shortcomings and omissions:

1. In order to access the "knowledge" component of these hybrid systems, the entire CAD system must always be up and running concurrently with the KBE add-on. This results in a significant overhead and cost.

2. These augmented CAD systems are not, by their nature, inherently web-based frameworks.

3. They are supported with a rather narrow choice of computer and Operating System platforms. Notable by its absence is Linux, an increasingly important player these days, especially for server-based applications.

4. In general, they do not provide full generative modeling, including dynamic typing and consistent, system-wide caching and dependency-tracking, all of which are key features of a bonafide KBE system.

5. These CAD augmentations are not, at their core, language-oriented systems. If there is an embedded underlying representation language present, it is hidden from not only the end-user, but from the developer as well. This results in the engineer being constrained to work in a universe of a finite number of menu clicks.

6. The Modeling language is "interpreted," as contrasted to being compiled. This can result in a tremendous disparity in the speed of developing and running an application. This commonly not understood difference becomes crucial as KBE applications enter the "enterprise," mission-critical arena.

7. In order to achieve the levels of customization needed for even a moderately complex implementation, these systems invariably require a link into the innards of the CAD systems API, or Application Programming Interface. With Catia, for example, this requires adding the extremely expensive CAA to the mix. Programming in the CAD API requires hard-core software development in yet another completely different environment (typically C++-based). Ironically, this ends up largely negating the very purpose for having a high-level KBE language for engineers in the first place.

[4]GDL uses SMLib from Solid Modeling Solutions, http://www.smlib.com

LANGUAGE-BASED KBE SYSTEMS These are the "standalone" systems in the sense that they do not require a desktop CAD system to host them while running. These systems fall into two major categories: Those based entirely on proprietary languages, and those based on Industry Standard languages.

Proprietary Language KBE Proprietary-language KBE tools typically utilize a generative language, with accompanying caching and dependency tracking. They contain a home-grown compiler or interpreter. They may resemble a subset of some standard language such as C or Java, but they do not comprise a superset of any accepted standard foundation.

From our perspective, the use of a proprietary tool has at least two critical downsides:

- The user is locked into the single vendor – that is, the user can only "go shopping" at the Company Store. The Proprietary nature of the code is inconsistent with the users' ability to bolt on other tools (i.e. libraries) that are open source or standards-based. In today's marketplace this has become increasingly important, to the point of being a must-have capability.
- This nonstandard nature creates the potential of risk when upgrading to new versions of the tool. Since there is no Industry Standard governing how the core system must behave, new versions inherently pose the potential risk of breaking existing applications and libraries.

Standards-based Language KBE Standards-based Language KBE systems, as exemplified by Genworks GDL (General-purpose Declarative Language), provide a pure, standalone, language-Based KBE solution. Because GDL is a superset of ANSI Common Lisp:

- There is a choice of vendors. Genworks currently ships with Allegro CL from Franz Inc or Lispworks from Lispworks Ltd, and can potentially use any number of other available ANSI CL implementations, both commercial and open-source.
- The Genworks GDL language allows any number of available open-source CL libraries to be plugged in. Some of these are already provided by Genworks with its standard GDL package, and many others are available from common-lisp.net and other sources. At last count, common-lisp.net lists 347 free open-source libraries for wide-ranging applications from matrix processing to text file parsing to genetic optimization algorithms.

In addition to its open-source capabilities, GDL provides the following benefits, and can do so in large part because it is built on a mature, standards-based foundation:

- The Geometry is provided by the Solid Modeling Solutions Company, makers of what reviewers have described as the premier independent geometry kernel, SMLib. It too is fully standards-compliant in the areas of NURBS routines and geometry exchange.
- As delivered, GDL does not depend on any other "engines" to be tacked on.
- GDL is fully web-centric, both as to run-time applications and the development environment itself. This, without exception, makes the development and deployment cycle appreciably speedier than that of other systems in any of the categories.
- GDL is cross-platform and supported on Linux, Windows, Macintosh and Commercial Unix.
- GDL is fully and dynamically generative. That is, objects can change their type, their cardinality, and have their tree structure altered, in a dynamic fashion, both while being programmed and at run time.
- GDL contains high-performance state-of-the-art Dependency Tracking, a crucial core feature of any KBE system.
- GDL is a fully Language-Based technology, drawing upon cutting edge macroexpansion technology. This provides a totally consistent environment and extremely extensible capabilities.
- The GDL Modeling language is constantly being compiled. The Franz Allegro CL compiler is the result of decades of compiler design and optimization by experts in that field, and supported by Franz.

EXAMPLE APPLICATIONS

DELFT MULTI MODEL GENERATOR Delft University of Technology's Multi-Model Generator (MMG) project (figures 6 and 5) makes use of multidisciplinary methods for generating a wide range of aerodynamic shapes for aircraft, rotorcraft, airships, and turbine blades. The methods used include the CST method for airfoil geometry mathematics(8) from Brenda Kulfan of the Boeing Company. CST offers some elegant and simple formulas for generating a wide variety of airfoil shapes – formulas which fall outside the standard CAD toolbox of NURBS, but which represent airfoil shapes more naturally.

Using GDL, the Delft researchers are able to automate the CST approach, as well as conveniently integrate it with a standard CAD-friendly NURBS-based modeling system and put the resulting airfoils through optimization with simple integration to standard tools like X-foil. The following code snippet captures the basic class function offered by CST. It is simply instantiated by the GDL application whenever a set of sample points are to be put through this class function. Then a standard NURBS curve can be fitted to the sampled points, to a specified tolerance.

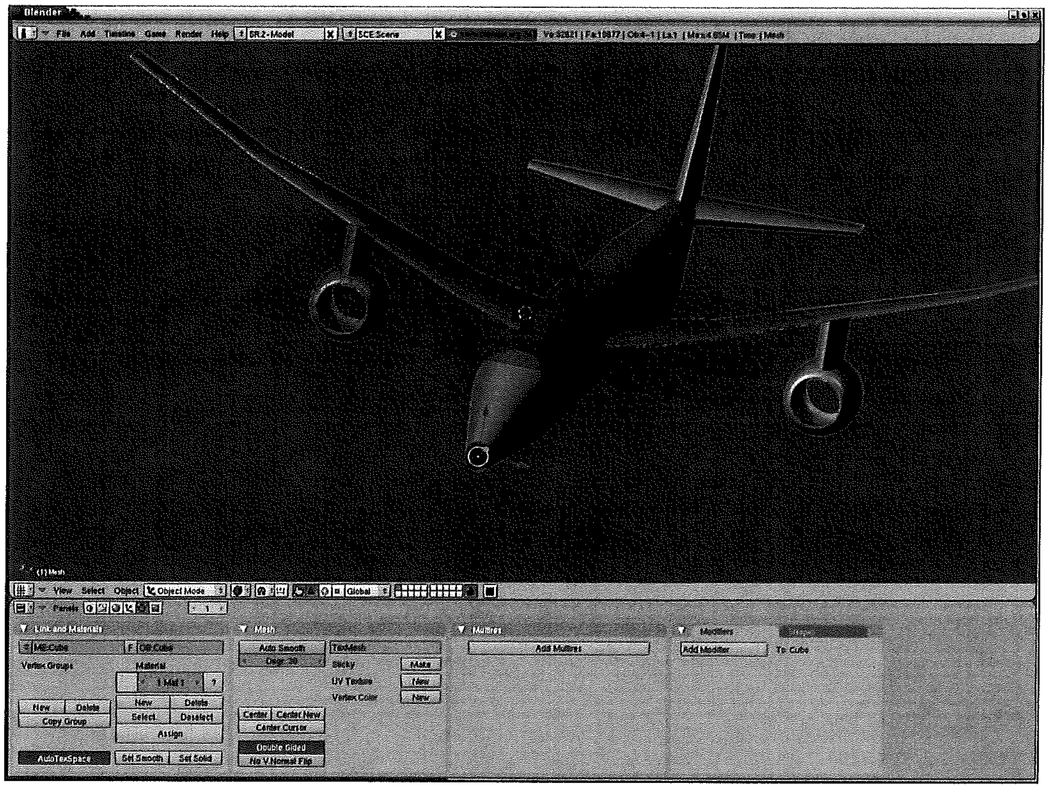
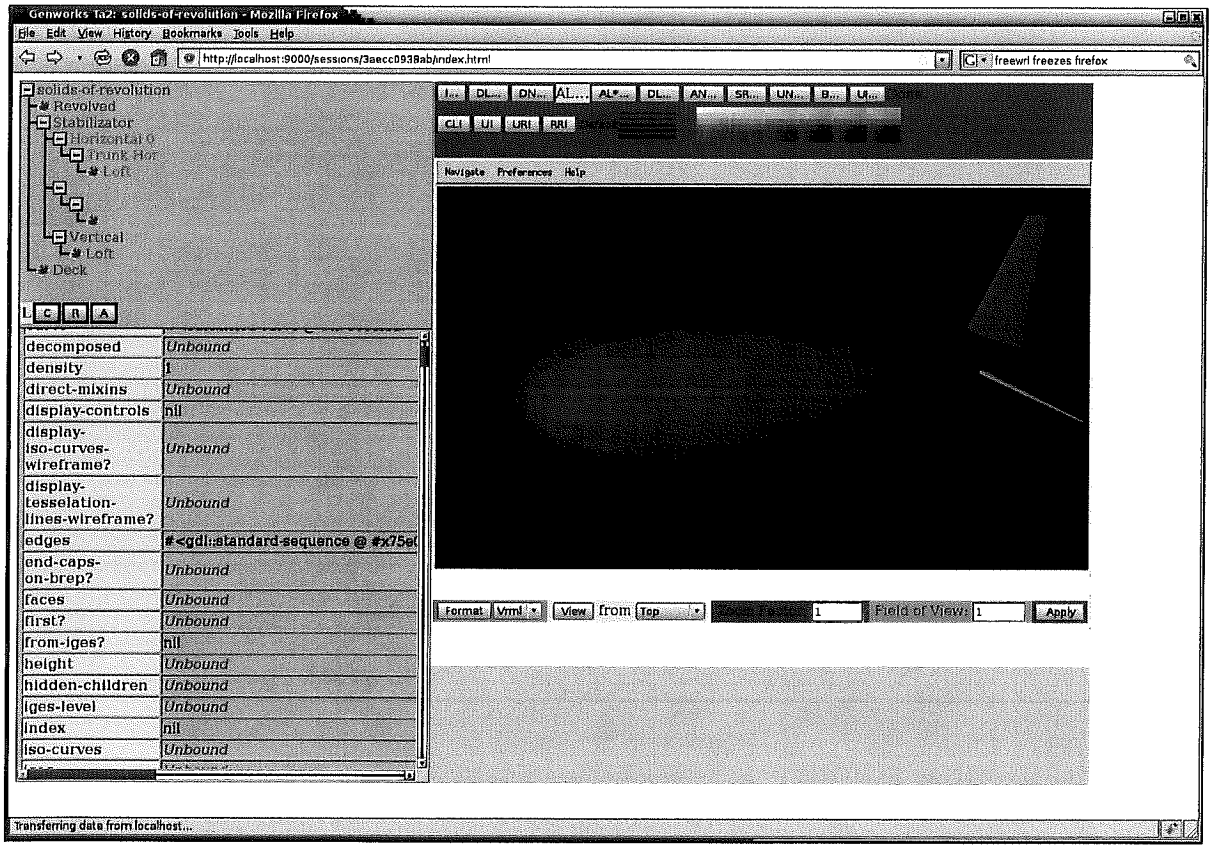


Figure 2: STL model of aircraft, being processed by Blender3D


Genworks Ta2: solids-of-revolution - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:9000/sessions/3aecc0938ab/index.html
freewrl freezes firefox
solids-of-revolution
Revolved
Stabilizator
Horizontal 0
Trunk Hor
Loft
Vertical
Loft
Deck
L C R A
Navigate Preferences Help
Format Vrml View from Top Field of View: 1 Apply
Transferring data from localhost...

decomposed	Unbound
density	1
direct-mixins	Unbound
display-controls	nil
display-iso-curves-wireframe?	Unbound
display-tesselation-lines-wireframe?	Unbound
edges	#<gdl::standard-sequence @ #x75e(
end-caps-on-brep?	Unbound
faces	Unbound
first?	Unbound
from-iges?	nil
height	Unbound
hidden-children	Unbound
iges-level	Unbound
index	nil
iso-curves	Unbound

Figure 3: VRML rendering of airship, in Ta2 Development Browser running in Firefox 3

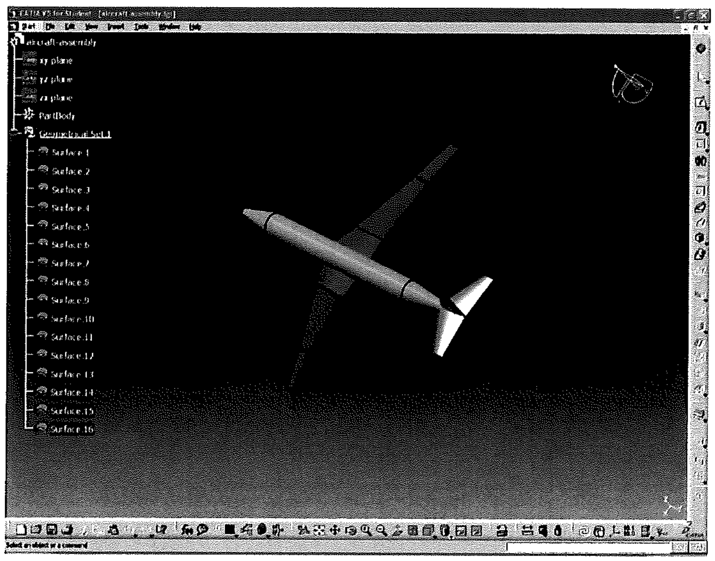


Figure 4: Aircraft model delivered into the CATIA environment

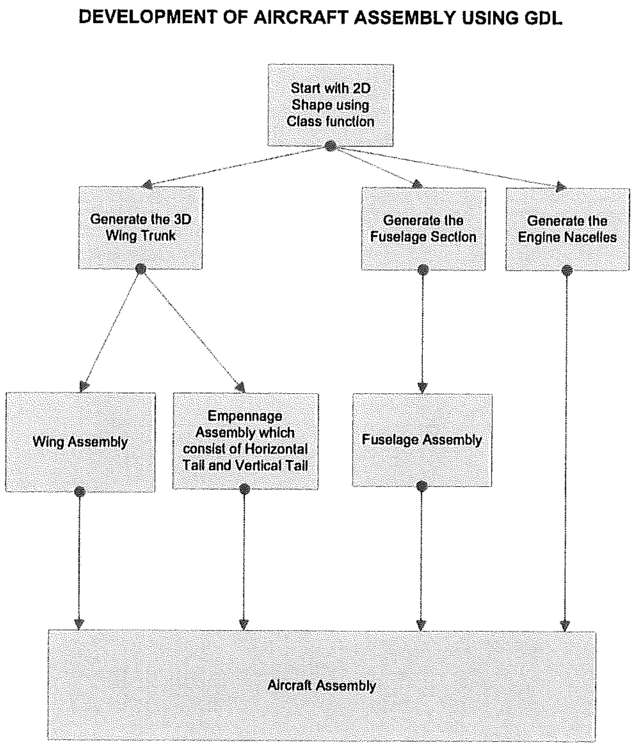

DEVELOPMENT OF AIRCRAFT ASSEMBLY USING GDL
Start with 2D Shape using Class function
Generate the 3D Wing Trunk
Generate the Fuselage Section
Generate the Engine Nacelles
Wing Assembly
Empennage Assembly which consist of Horizontal Tail and Vertical Tail
Fuselage Assembly
Aircraft Assembly

Figure 5: Process Flow for GDL Aircraft Design Generation

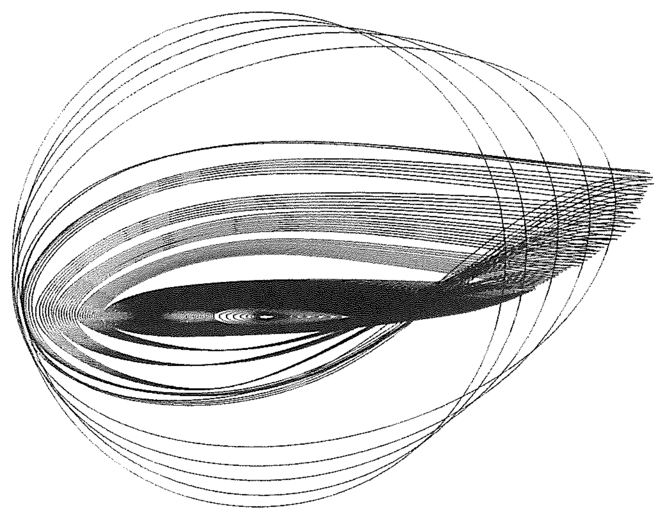


Figure 6: Generating Initial Profiles for Wind Turbine Blade

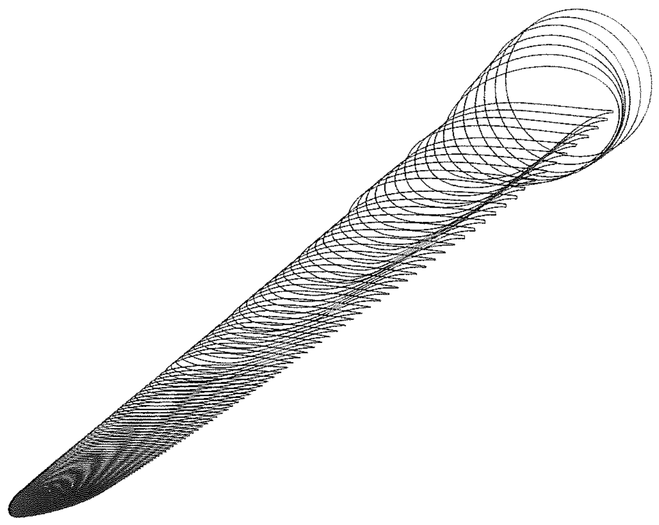


Figure 7: Wind Turbine blade profiles positioned and oriented for use by surface skinning routine

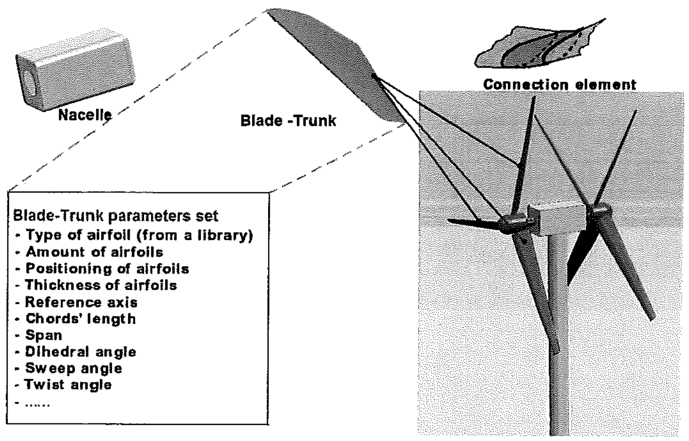


Figure 8: Process Flow for GDL Wind Turbine Design Generation

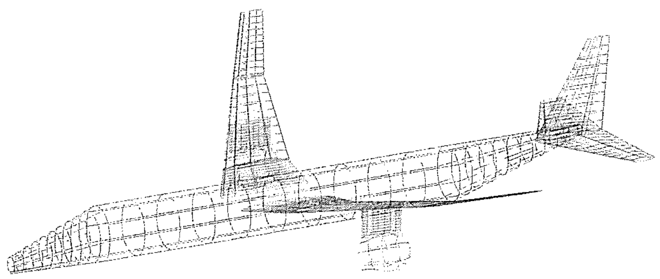


Figure 9: example PDF output (vector graphics with infinite resolution)

```
(define-object class-function (base-object)

  :input-slots
  ((n1 .5 :settable)
   (n2 1 :settable)
   (step-size 0.01))

  :computed-slots
  ((x-list (list-of-numbers 0 1 (the step-size)))

   (y-list (mapcar #'(lambda(psi)
                       (* (expt psi (the n1))
                          (expt (- 1 psi) (the n2))))
                   (the x-list)))

   (upper-points (mapcar #'(lambda(x y) (make-point x y 0))
                   (the x-list)
                   (the y-list)))

   (lower-points (reverse
                  (mapcar #'(lambda(point)
                              (make-point (get-x point)
                                          (- (get-y point))
                                          (get-z point)))
                          (the upper-points))))

   (number-of-points (length (the upper-points))))

  :objects
  ((polyline :type 'global-polyline
             :vertex-list (append (the upper-points)
                                  (the lower-points)))))
```

Figure 10: The CST captured in GDL

```
(define-object nacelles (base-object)

  ...

  :objects
  ((revolved :type 'revolved-surface
             :curve (the composed)
             :axis-vector (the (face-normal-vector :front))))

  :hidden-objects
  ((composed :type 'composed-curve
             :curves (list (the fit) (the trailing-edge-joint)))

   (fit :type 'fitted-curve
        :points (the points-list)
        :tolerance 0.01)

   (trailing-edge-joint :type 'linear-curve
                        :start (first (the points-list))
                        :end (lastcar (the points-list)))

   (class-function :type 'class-function
                   :step-size (the class-function-step-size)
                   :pass-down (n1 n2)))
  ... )
```

Figure 11: The Class Function being used in a Nacelle object

In the code in Figure 10, the repeated use of `(mapcar #'(lambda(...) ...) ...)` is a simple way to apply a function (the lambda expression) to each element in a sequence or list. In this case, this mapping often applies to a list of points which are being "pumped" through a transformation function.

In 11, we see how the class function is incorporated very easily into the declarative model. Just as we "call" a function in a procedural language, we can instantiate an object in a KBE language. Input values are specified and demanded when needed, and the output from the object (in this case the Class Function object) are demanded, computed, and cached when they are needed.

With a small additional amount of similar code, the Delft MMG is able to generate full models as seen in figures 9 and 3.

When the researchers have developed a model definition, assembling it from a combination of High-level Primitives (HLPs) and custom object definitions unique to the particular model, they then have a completely flexible and automatable model for use in Multi-disciplinary Optimization and Analysis.

Because the models are developed in a superset of a rich, mature, ANSI standard language environment, there are no arbitrary limits on automating tasks such as:

- Generating thousands of model instances, and sampling the computed results, with the input parameters controlled by optimization algorithms, catalog databases, etc;
- Preparing output in virtually any target format, for use by outside tools;
- Reading input from virtually any source format, for use in generating new model instances.

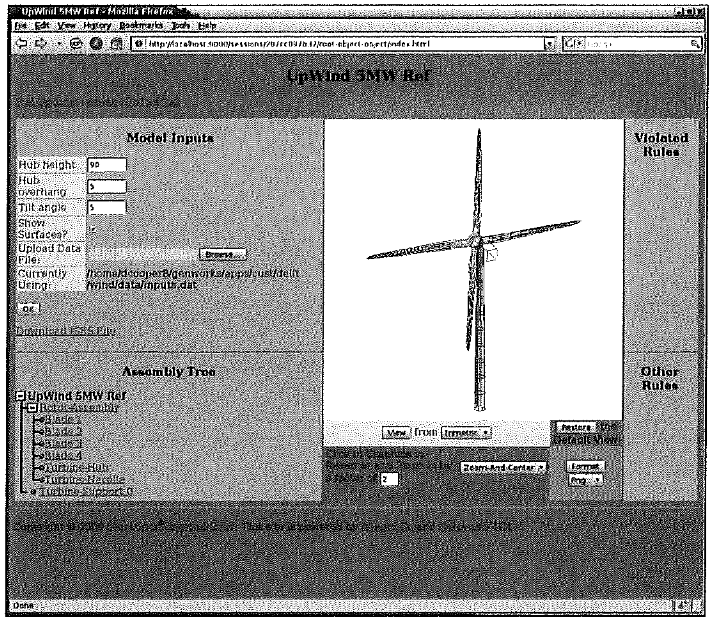
This level of flexibility is unique to a language-based KBE system.

TA2 DEVELOPMENT BROWSER While developing a KBE application, it is important for the developer to receive constant feedback. While much of this feedback can come from the rich editing environment itself (e.g. Emacs), the ultimate goal is usually to develop an end-user application which runs on the web.

For this purpose, Genworks provides a generic Web application, "ta2" which runs on any standard web browser and can be used for testing, tracking, and visualizing the results of any KBE application being modeled in the system. The beauty of this approach is that the development environment is essentially the same as the runtime deployment environment. Ta2 itself is really just another GDL web application.

Therefore all the same benefits which apply to deployed runtime applications also apply to using the ta2 development environment:

- Runs identically regardless of computer platform;
- Network-friendly deployment (ta2 can be run from a remote machine on the network).
- It takes advantage of standard CSS style sheets and other web enhancements (ta2 is currently being enhanced with visually-appealing CSS style sheet design).
- link directly to online documentation, also available through the web;
- Instantly test the end-user web interface, running side-by-side with the ta2 application.


UpWind 5MW Ref - Mozilla Firefox

UpWind 5MW Ref

Model Inputs

Hub height: 90
Hub overhang: 5
Tilt angle: 5
Show Surfaces?
Upload Data File:
Currently Using: /home/dcooper8/genworks/apps/cust/delft/wind/data/inputs.dat

Assembly Tree

UpWind 5MW Ref
- Rotor-Assembly
  - Blade 1
  - Blade 2
  - Blade 3
  - Blade 4
  - Turbine-Hub
  - Turbine-Nacelle
- Turbine-Support-0

Figure 13: Wind Turbine skeleton user interface, accessed directly from ta2

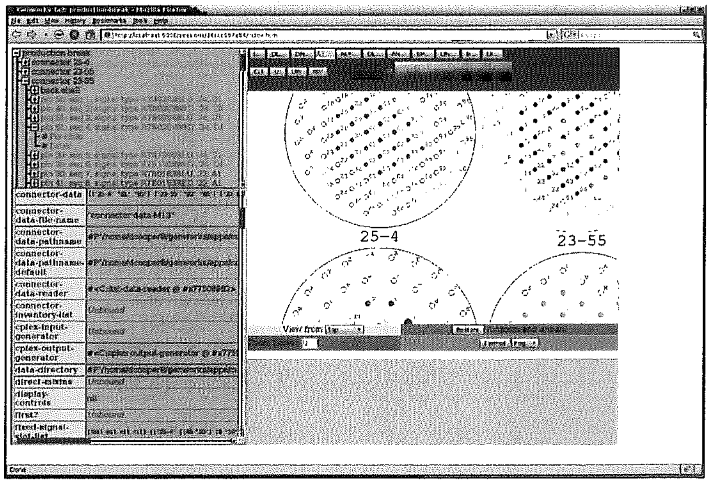


Figure 14: ta2 environment in Firefox with aircraft wiring connector model. Model tree, object inspector, and graphics viewport are currently visible

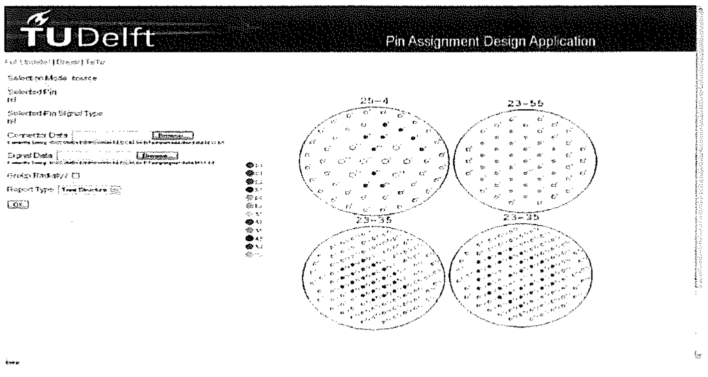
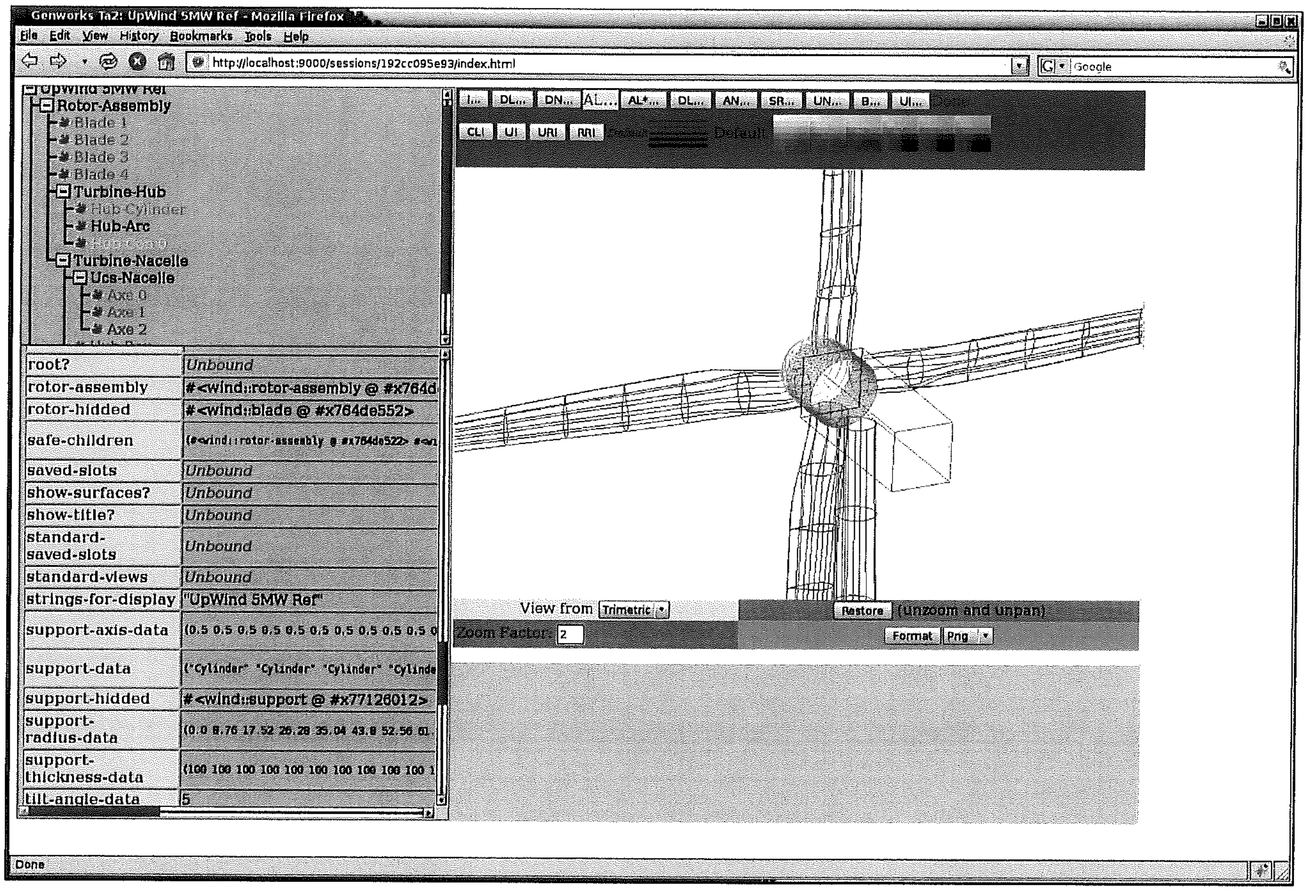


Figure 15: Aircraft Wiring connector model user interface, accessed directly from ta2



root?	*Unbound*
rotor-assembly	#<wind::rotor-assembly @ #x764d
rotor-hidded	#<wind::blade @ #x764de552>
safe-children	(#<wind::rotor-assembly @ #x764de522> #<wi
saved-slots	*Unbound*
show-surfaces?	*Unbound*
show-title?	*Unbound*
standard-saved-slots	*Unbound*
standard-views	*Unbound*
strings-for-display	"UpWind 5MW Ref"
support-axis-data	(0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0
support-data	("Cylinder" "Cylinder" "Cylinder" "Cylinde
support-hidded	#<wind::support @ #x77126012>
support-radius-data	(0.0 8.76 17.52 26.28 35.04 43.8 52.56 61.
support-thickness-data	(100 100 100 100 100 100 100 100 100 100 1
tilt-angle-data	5

Figure 12: ta2 environment in Firefox with wind turbine model. Model tree, object inspector, and graphics viewport are currently visible

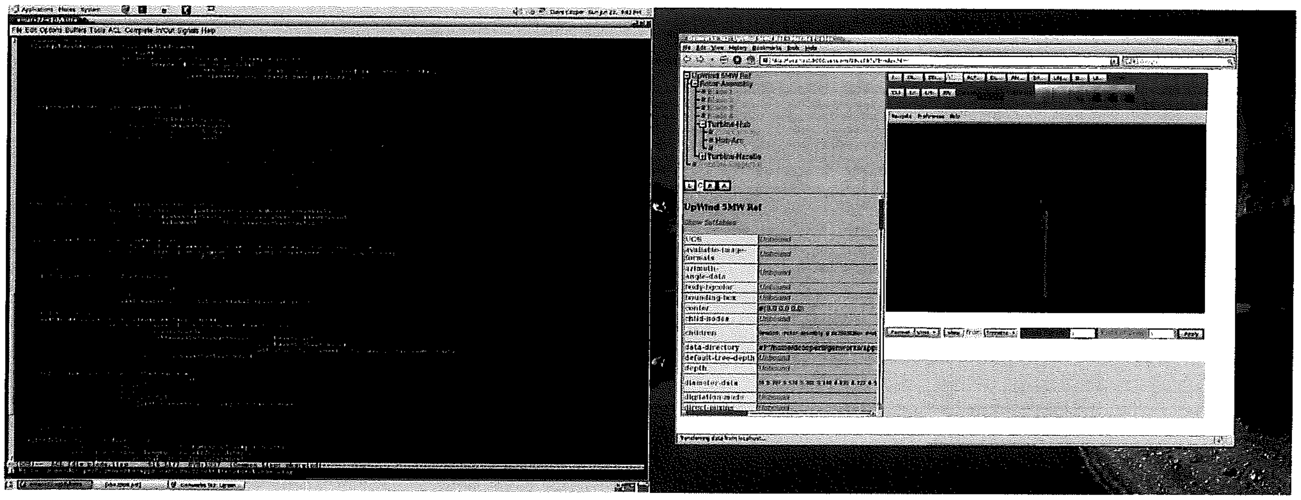


Figure 16: Emacs and ta2 in Firefox side-by-side in dual display, typical KBE development session

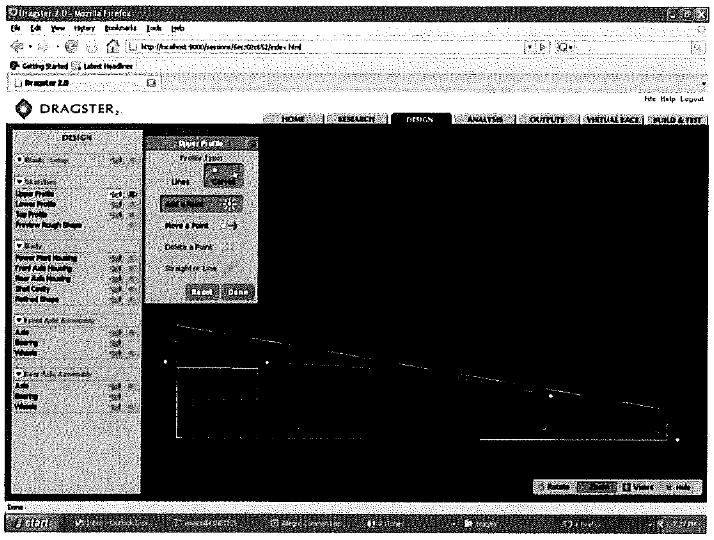


Figure 17: Whitebox Dragster GDL-based 2D Sketcher, running in Firefox web browser

In summary, ta2 is a KBE application whose purpose is developing other KBE applications. This kind of "recursive" application of KBE technology is a typical KBE approach.

DRAGSTER 2.0 The Whitebox Dragster is a Web 2.0 application which has been developed in GDL and makes use of cutting-edge Web3D technology. The Dragster application has been able to capitalize on the flexibility of GDL to provide a dynamic and high-quality 3D graphical user experience, all running from within a standard web browser and without any application software installed on the user's computer.

From the website of Whitebox Learning:

> The WhiteBox Dragster system complements the popular CO2 car racing competition that takes place in thousands of schools each year. By providing students with extensive theoretical fundamentals, and most importantly, the tools necessary to make this theory actionable, WhiteBox Dragster takes this great learning activity to an entirely new and exciting level.

Originally the Whitebox system was developed in a proprietary-language based KBE tool which required heavyweight desktop software to be installed on the end user's computer. However, in practice, this proved to be too unwieldy an approach for large-scale deployment.

Fortunately, Whitebox Learning has been able to use Genworks GDL to resurrect the Dragster, and make it shine as a pure web-based application.

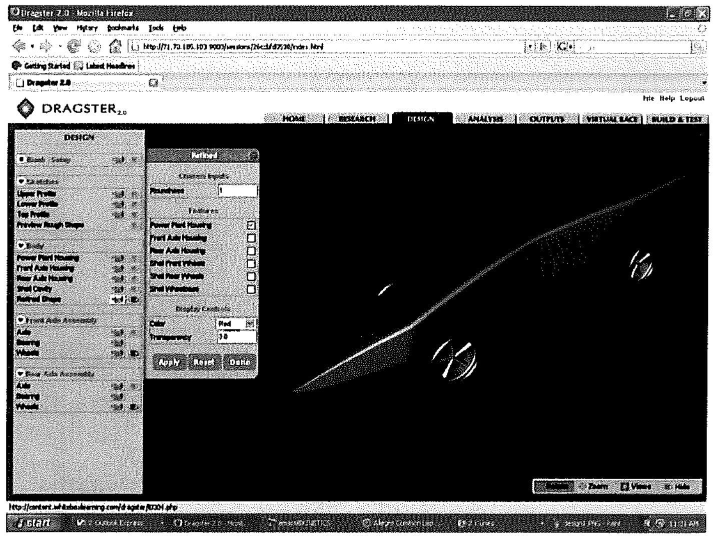


Figure 18: Whitebox Dragster GDL-based 3D model refinement, running in Firefox web browser

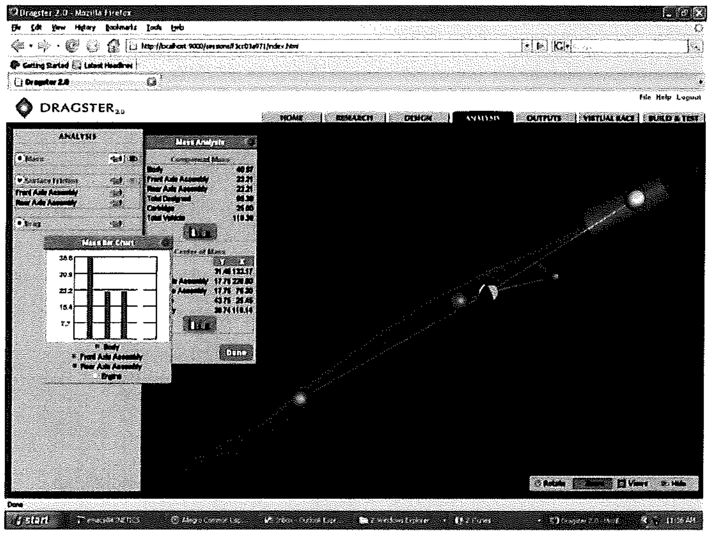


Figure 19: Whitebox Dragster GDL-based 3D model analysis, running in Firefox web browser

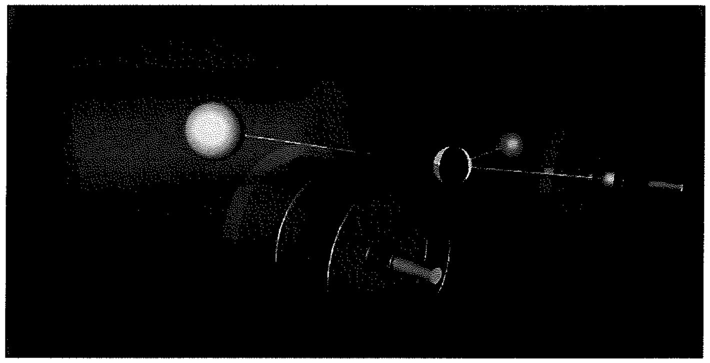


Figure 20: Whitebox Dragster GDL-based perspective 3D view, running in Firefox web browser

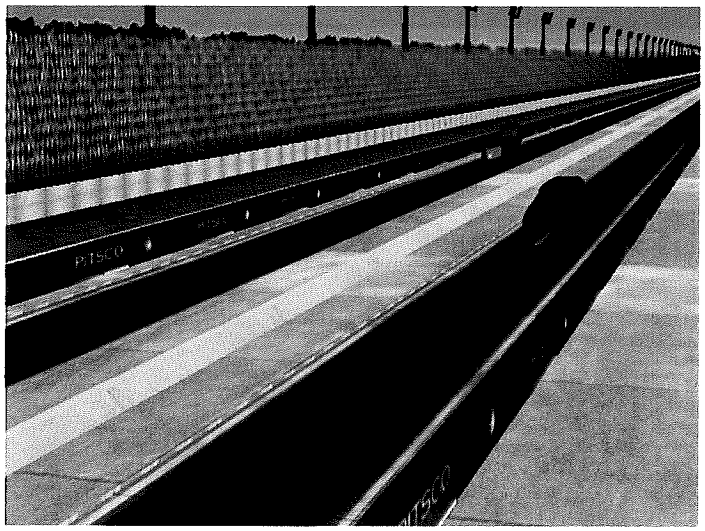


Figure 21: Whitebox Dragster GDL-generated Web3D Virtual Race, running in Firefox web browser

CONCLUSION

The over-arching goal of KBE in this century is to enable the power of modern computing to come together with the creativity in the mind of an engineer, and to do so in the most effective ways possible. Over the years, this vision has taken on many forms, but certain recurrent themes have emerged as the "keys to success" for a durable KBE presence. Those keys at bedrock are: simplicity of the language syntax, ease of use, and a commitment to build on existing technology and Industry Standards wherever possible.

REFERENCES

[1] Switlik, J. M., Wikipedia article on KBE (http://www.wikipedia.org/knowledge_based_engineering), Nov. 2006.

[2] Premetz, G., COE Newsletter, Nov.-Dec. 2003.

[3] Cooper, D. and van Tooren, M., "Knowledge-based Techniques for Developing Engineering Applications in the 21st Century," *AIAA/ATIO Conference, Belfast, October 2007*, 2007.

[4] Nicolaescu, D., http://groups.google.com/group/gnu.emacs.sources/msg/a188ece1a448b031, 2007.

[5] Gorrie, L., http://common-lisp.net/project/slime/, 2006.

[6] Abelson, H. and Sussman, G. J., *Structure and Interpretation of Computer Programs*, The MIT Press, 1996.

[7] Sussman, G. J. and Wisdom, J., *Structure and Interpretation of Classical Mechanics*, The MIT Press, 2001.

[8] Kulfan, B. M., "Recent Extensions and Applications of the CST Universal Parametric Geometry Representation Method," *AIAA/ATIO Conference, Belfast, October 2007*, Oct. 2007.

[9] LaRocca, G., *Knowledge Based Engineering Techniques to Support Aircraft Design And Multidisciplinary Analysis and Optimisation (DRAFT)*, Ph.D. thesis, Technical University of Delft, South Holland, Netherlands, Aug. 2007.

[10] La Rocca, G. and van Tooren, M., "Enabling Distributed Multi-Disciplinary Design of Complex Products: a Knowledge Based Engineering Approach," *J. Design Research*, Vol. 5, No. 3, 2007, pp. 333–352.

CONTACT

Contact with the principal author, Dave Cooper, may be had through email at david.cooper@genworks.com. The Genworks International website is: http://www.genworks.com.

ADDITIONAL SOURCES

The following is directly derived from Section 1.6 of Gianfranco LaRocca's Draft PhD Thesis(9) at Technical University of Delft. Gianfranco used ICAD as the representative example; the version here is modified slightly to use Genworks GDL.

The most outstanding example of a macro provided by various KBE systems (though in different forms) is the one used for defining classes and hierarchies of objects. Mastering the use of such a macro is fundamental for developing any KBE application. As a representative case, the GDL-specific macro `define-object` is discussed here, as GDL is the most familiar KBE system to the author. In fact, ICAD, Knowledge Fusion (KF), and others provide their own version of a similar construct, though different names and slightly different syntax are used. As matter of fact, it was possible to construct a simple mapping table among ICAD, GDL, and KF constructs, which covers a sampling of common usage and will allow the interested reader to understand the structure of ICAD defpart and KF defclass, on the basis of the GDL define-object (see the original LaRocca paper for this table(10)[5]).

The `define-object` macro (or the non-GDL equivalent) is the basic means to apply the object-oriented paradigm in KBE applications. It allows defining classes, superclasses, objects and relationships of inheritance, aggregation and association, as discussed [elsewhere in the LaRocca dissertation]. The `define-object` macro is basically structured as follows:

[5]Automatic conversion/translation among KBE language formats is also quite possible. For example, GDL has a simple open-source module available to convert legacy ICAD models into GDL "on-the-fly" and compile them into usable definitions, transparently to the user.

THE MAIN SECTIONS OF AN OBJECT DEFINITION

Name of the class: this is a user-chosen symbol which names the object definition.

Mixin-list: this is a list of superclasses or other classes from which the class here specified will inherit all the characteristics (attributes and components). The classes specified here can either be formal superclasses (of which the class specified in the define-object is an actual specialization), or just other classes with which this class is to share attributes and parts (see next items).

Input-slots: this is a list of parameters to be assigned in order to generate an instantiation of the given class. This set of parameters represents the so-called "class protocol." Default values can be specified outside the protocol.

Computed-slots: This is a list of lists; each internal list contains a slot name, a value or expression, and optionally one or more modifier keywords. The value or return-value of the expression becomes the slot's value after it is computed on-demand. These expressions can either be production rules or any other mathematical, logic or engineering rule.

Objects: this is the list of objects which may be contained in an instance of the `define-object`. They are also called "children" of the `define-object` instance. For each object, the following must be specified:

- the object name
- the name of the relative class to be instantiated (by using the keyword `:type`)
- values for the input-slots required for the instantiation. This parameter list must sufficiently match the protocol of the class to be instantiated (i.e. at least its required input-slots).

Functions: these are similar to `computed-slots`, but, like functions in raw Lisp, they can accept arguments, and their computed return-values are not cached as is done by default with computed-slots.

MESSAGES, *THE* REFERENCING, AND REFERENCE CHAINING Input-slots, computed-slots, objects, and functions are all considered *messages* of the object, and all return a deterministic value when computed. To compute these expressions it is possible to use and combine values of other messages, either messages defined locally in the object definition, or ones inherited by the classes specified in the mixin-list. This is done with *the* referencing, e.g. `the length` will return the value of the `length` message, however it is defined in an object.

With *reference chaining*, it is possible to use slots of the child object or other descendant or ancestor objects in the instantiated object tree defined in the given `define-object` form.

These are the fundamental elements typical of KBE systems. Beyond these basics, more advanced constructs are available to facilitate and organize particular activities, for example:

- Functions and Methods for traditional procedural, functional coding style;
- Outputting data and geometry in various formats;
- Constructing user interfaces for environments such as web browsers