Knowledge based engineering: Between AI and CAD. Review of a language based technology to support engineering design

Gianfranco La Rocca

Faculty of Aerospace Engineering, Delft University of Technology, Chair of Flight Performance and Propulsion, Kluyverweg 1, 2629HS Delft, The Netherlands

1. Introduction

Knowledge based engineering (KBE) stands at the cross point of diverse fundamental disciplines, such as artificial intelligence (AI), computer aided design (CAD) and computer programming. Though these individual contributing disciplines are widely represented in scientific literature, KBE as of yet, is not. To date, no scientific books can be found that are dedicated to this topic. This can be explained by the fact that KBE has been for many years the exclusive domain of a few and highly competitive industries (aerospace and automotive in particular) and has not entered mainstream academic research.

The limited amount of available information, mainly in form of pamphlets from KBE vendors, has not stimulated the scientific community interest in KBE as a real engineering discipline. In addition, it has always been difficult to precisely position this technology within the panorama of scientific research. Is KBE about CAD development? Or is it about artificial intelligence? Or IT? Eventually, this mix of inherent ambiguity and scarce information - possibly encouraged by an inconvenient name: what kind of engineering is not based on knowledge! – has marked the difficult story of KBE to date.

At the beginning of 2000, a 20 page booklet entitled Achieving Competitive Advantage through Knowledge-Based Engineering [1] was prepared for the British Department of Trade and Industry, with the stated aim of demystifying the terminology surrounding KBE and to explain how such a technology could be deployed to gain competitive advantage. In the format of a best practice guide, it starts with a very brief and high level description of the fundamental concepts of KBE, then focuses on the possible impact of KBE on the business and concludes with some implementation and use guidelines. Although acknowledged to be one of the most popular readings on KBE, this document did not offer sufficient technical content to spark the interest of the scientific community.

Whilst the strengths and opportunities of using KBE are well highlighted, no detailed information is given about the technology underpinning KBE, neither indications on the sort of activities involved in the development of a KBE application. In the same period, two journal papers by Chapman and Pinfold were published, offering a much more scientific look at KBE [2,3]. To date they are possibly the two most cited publications in the field of KBE. Yet, the main focus of those papers was on the specific application cases rather than KBE technology.

So far, several other design cases claiming and demonstrating the potential of KBE in the field of engineering design have been published in scientific literature [4–10], but, still, almost nothing that focuses just on KBE technology. The purpose of this paper is to fill this gap and offer a review of the technological fundamentals of knowledge based engineering. Although some references and a few examples will be provided, no specific application and validation cases are thoroughly discussed in this paper.
Knowledge based engineering technology can be positioned in the group of so called knowledge technologies. In Ref. [11], Milton makes use of this term to address “a set of new computer-based techniques and tools that provide a richer and more intelligent use of information technology”. The power of knowledge technologies comes from the way they combine ideas and applications from a surprising broad and heterogeneous set of fields, including psychology, philosophy, artificial intelligence, engineering, business management, computer science and web technologies. The common denominator of knowledge technologies is their focus on knowledge and its management: for example, how to identify relevant knowledge in an organization; how to capture and formalize it for more efficient reuse; how to represent and store it to improve access, maintenance and transfer; how to embed it in computer systems to provide benefit. The development of computer systems that help engineers to increase the efficiency of their work by enhancing the level of automation in the design process, is definitely the area of interest of KBE. Knowledge capture, representation, retrieval, as well as knowledge coding and inference are some aspects which are definitely related to KBE and the development of KBE applications; however they fall in the focus areas of other contiguous disciplines, such as knowledge engineering and knowledge management.

The boundaries and categorization of these disciplines and their relative (knowledge) technologies are quite fuzzy and subjective. In Ref.[12], McMahon et al. position KBE in the field of knowledge management, together with other knowledge technologies, such as data mining and ontology engineering, as well as with those technologies for computer-supported collaborative work that range from email to desktop sharing systems and video conferencing. Within the Airbus company, one of the pioneering organizations in the adoption of KBE, and, similarly, inside the Dutch aerospace companies Fokker Elmo and Fokker Aerostructures, KBE is considered one technology in the broader field of knowledge engineering.

According to the typical methodological approach to practice KBE, before starting with the development of any KBE application, it is required to proceed, first, with the identification, then the acquisition and, finally, the codification of the relevant knowledge that will have to be embedded in the KBE application itself. Once developed, the given KBE application will be deployed, typically as part of a broader and heterogeneous engineering design framework, where it will be integrated with other computer aided engineering tools by means of some workflow management system. Already in this oversimplified description of the development and deployment process of a KBE application, it appears that the interest areas of KBE, Knowledge Engineering and knowledge management intersect, complement and specialize each other. While in the KBE area, the focus is on the technical development of the KBE application, in the knowledge engineering area, the emphasis is on the acquisition and codification of knowledge. Within the knowledge management area the attention is on the overall goal of nurturing and supporting initiatives that can enable a more efficient and effective use of the knowledge assets in the organization. In each of the three areas, specific tools and (knowledge) technologies are developed and brought in use. A graphical representation of this context is attempted in Fig. 1.

Aspects not strictly related to KBE and its technological fundamentals will not be treated further in this paper. However, the reader is referred to the work of Shreiber et al. [13] to learn more about methodologies to support the development of generic knowledge based systems. The reader interested in the development of methodologies specific for KBE systems, is advised to refer to the work generated in the framework of the MOKA project [14–16]. An overview on a broad set of knowledge technologies can be found in Ref.[11]. Here, apart from KBE, a friendly introduction is provided to semantic technologies and ontology languages to codify and exchange knowledge, such as KIF (knowledge interchange format), RDF (resource description framework) and OWL (ontology web language). Ref. [17] offers a step-to-step guide to knowledge acquisition, which falls outside the area of KBE, but represents an essential step toward the development of any KBE application.

In synthesis, this paper is not going to discuss when KBE should be used, who should use it (further interesting readings in Ref.[18–20]), and how it should be used, but it will focus on what KBE is and what developing a KBE application is about, where KBE is coming from, and, to conclude, where KBE is and should be going.

To achieve these aims, the paper is structured as follows. To begin, a comprehensive definition of knowledge based engineering is provided in Section 2. In Sections 3–4 the origins of KBE are investigated. The similarities and main differences with its AI ancestors are then discussed in Section 5. Section 6 represents the main body of the paper and deals with the programming approach of KBE. Indeed, the use of a programming language is the main characteristic of state-of-the-art KBE systems. The typical nature and main features of KBE languages are covered in detail. Sections 7–8 provide a categorization of the design rules that can be manipulated in a KBE system to perform generative design. Section 9 collects some reflections on the convenience of the programming language approach to support engineering design and the relative positioning of KBE and CAD in the engineering design process. Section 10 describes the evolution and the current trends in KBE technology, thereby covering the major advances in the field since the time of the first best practice guide mentioned above. Section 11 provides a number of conclusions, together with a list of recommendations and expectations for the future development of KBE.

2. Knowledge based engineering: a definition

Various definitions of knowledge based engineering can be found in literature, which typically reflect the different views held by differing KBE stakeholders. A company manager can see KBE as a technology asset to compress product development time and cut engineering costs. A KBE developer, i.e. the user of a KBE system for the development of KBE applications, sees it as a refined type of software development tool incorporating aspects of object oriented (OO) and functional programming. Engineers and designers, i.e. the typical users of KBE applications, might see KBE as a technology to
augment the dynamic calculation capabilities of classical spreadsheets with generative geometry and report-writing capabilities, or, vice versa, to augment the level of automation and “intelligence” of conventional CAD systems by embedding design rules and engineering knowledge. Eventually, engineers would see KBE as a technology that can support their work better than other conventional IT technologies (e.g., spreadsheets and CAD) through the automation of routine design tasks and easier implementation of multidisciplinary design optimization (MDO) methodology. Knowledge management practitioners can see KBE as one technological solution in their portfolio to retain company knowledge and support its effective and systematic reuse. Actually, KBE is all of this, which leads to the author’s extended definition [21]:

Knowledge based engineering (KBE) is a technology based on the use of dedicated software tools called KBE systems, which are able to capture and systematically reuse product and process engineering knowledge, with the final goal of reducing time and costs of product development by means of the following:
- Automation of repetitive and non-creative design tasks
- Support of multidisciplinary design optimization in all the phases of the design process

3. The AI roots of knowledge based engineering

KBE is not a novel and revolutionary product from the world of computer science, but has strong roots in the field of artificial intelligence, particularly in the knowledge based systems (KBSs) technology of the 1970s.

Knowledge based systems (or expert systems) are computer applications that use stored knowledge for solving problems in a specific domain [22,23]. Similarly to a human expert, a KBS makes use of some kind of reasoning mechanism, the so called inference mechanism, to derive an answer to the posed problem. This is done on the basis of a limited amount of given facts (which specify the problem at hand), plus a body of domain specific knowledge, which has been previously stored in a dedicated repository, called knowledge base (KB).

To enable storage and accessibility by the inference mechanism, the domain knowledge needs to be structured and formalized by means of some symbolic representation. Rules and frames are the two most common forms of knowledge representation [11,22]. The first is based on the well-known IF-THEN construct. The second is a concept, proposed in 1970 by Minsky [24], based on the use of a collection of slots to describe attributes and/or operations featured by a given object. Each slot may contain a default value, a pointer to another frame, or a set of rules (including IF-THEN rules) and procedures by which the slot value is obtained. As simple example is shown in Fig. 2.

Eventually, these different symbolic representations have led to the definition of two types of KB systems, namely, the rule based systems (RBSs), which are possibly the most widely known, and the frame based systems (FBSs).

In RBSs the whole domain knowledge is expressed just as a collection of IF-THEN rules. In contrast, FBSs allow a much more sophisticated knowledge representation. Indeed FBSs are often seen as the application of the OO paradigm to the field of expert systems [22,25–27] and present classical OO features like abstraction and inheritance [25]. It follows that class-frames and instance-frames can be defined, where the latter are unique specifications of the generic class-frames, obtained through the assignment of a specific value set to the frame slots. Class-frames and superclass-frames can be defined, where the former are specializations of the first and, as such, inherit all the slots from the latter. These fundamental OO mechanisms allow FBSs to model both taxonomies and, since slots can contain pointers to other frames, part-whole aggregations.

To facilitate quick development of new knowledge based systems, special software tools, called shells, are available. These tools provide the basic components of a KB system, such as the knowledge base and the inference engine, without any specific domain knowledge inside. A shell can be thought as an expert system with an empty knowledge base and an interface to fill it. The initial idea behind shells was to reduce or eliminate the need of programming

Fig. 1. Relative positioning of knowledge based engineering, knowledge engineering and knowledge management. Lists of knowledge technologies involved in the various phases of the development process of a KBE application for engineering design.

Fig. 2. Simplified representation of a rule based system (RBS).
activities in the development of a KB system. However, it became clear that the bottleneck in KBS development was the knowledge acquisition and formalization phase, rather than the actual programming work [13,17].

Eventually, the ability to develop software applications able to mimic human experts and automate the problem solving process represented for the early KBE developers the same goal as for the expert systems developers of the 1970s. It is not a coincidence that a great deal of rule based systems and frame based systems genes have been inherited by current KBE systems. Of the two abovementioned types of KBS, frame based systems turned out to be much closer ancestors of knowledge based engineering technology, than the “simpler” rule based systems. In particular, the proven effectiveness of the frame concept in building complex knowledge representations can be found back in state of the art KBE systems, where new sorts of frames, with a different name and a significantly augmented set of capabilities, have been developed to address the specific challenges of the engineering design process.

4. Knowledge based systems + engineering = knowledge based engineering systems

Since the early 1970s, KBSs began penetrating the market of software applications, addressing problems of various complexity from different knowledge domains. “Help-on-line”, planning/scheduling systems and diagnostic tools represent just some of the commodity KBSs of daily use [11,23,28–30]. However, KBSs did not truly have an impact on the field of engineering design, including here aerospace, automotive and all those areas generally concerned with the development of complex hardware.

Apart from the inherent challenge of translating design knowledge into formal rules, the main reason for KBSs’ limited success was their inability to deal with two essential activities of the design engineering process, i.e., geometry manipulation and data processing. Indeed, KBSs are tools developed to solve problems by reasoning about facts and not really to perform computations and arbitrarily complex data processing tasks. Besides, KBSs do not have any competence in dealing with geometry and related shape configuration activity. On the contrary, most of the engineering design work requires and produces output that entails geometry manipulation, deals with the generation and management of complex products configurations, delivers data to various kinds of discipline-specific analysis tools and depends on the results of these analyses to advance the design process.

Specialized tools for geometry manipulation, data processing and computation in general proliferate in the engineering world. Those are the well-known computer aided design (CAD) systems and computer aided analysis (CAA) tools. The key question is whether a specific class of systems exists that can merge the capabilities of CAD and CAA systems with the reasoning competence and knowledge capturing and representation ability of KBSs.

These systems do exist: they are called knowledge based engineering systems and represent the evolution of knowledge based systems towards the specific needs of the engineering design domain. This explains also the origin of their name, which is a recurring source of dispute: it is just the merger of the terms knowledge based systems and engineering. KBE is just the technology behind these systems and there is no intended parsing of the term

Fig. 2. Examples of class and object frames; inheritance and part–whole relationships.
to indicate some other way of doing engineering, e.g., one which is not based on the use of knowledge...

KBE systems either have the capabilities of a CAD system built in, or are tightly integrated to an external CAD kernel (also a combination of the two). They allow both the implementation of analytical procedures (e.g., computational algorithms) and collaboration with external CAx tools. Finally, they are able to bear on a formalized body of knowledge to support rule-based design, hence relieving designers from the repetitive activities affecting the engineering design process.

Whenever a design case is highly rule-driven, multidisciplinary, repetitive and demands geometry manipulation and product (re)configuration, KBE is likely to be the best possible technology at hand [19,21].

As illustrated in Fig. 3 and often stated in literature [2], KBE systems can be considered as the merger of artificial intelligence and computer aided design technology. It is not by coincidence that two of the founding fathers of ICAD Inc., the company that, in 1984, brought the very first KBE tool (ICAD) to the market, were coming one from the AI laboratories of MIT and the other from the CAD company Computervision (then Parametrics and nowadays PTC) [31–33].

Whilst the entry of ICAD on the market can be considered as the very beginning of KBE, it cannot be considered the start of AI in CAD (or AI and CAD). At the beginning of the 1980s, a significant part of the international scientific community was already involved with the development of experimental systems to bring Knowledge Engineering capabilities into CAD [34]. It appears that the term intelligent CAD was already coined in 1983 [35,36], just to address this novel concept of CAD systems able to store knowledge and reason on it to support geometry generation.

The two epoch-making conferences organized in 1984 and 1987 on Knowledge Engineering in CAD and Expert Systems in CAD by the IFIP Working Group 5.21 were a clear evidence of the great activity in the field [37,38]. The scientific discussion on Intelligent CAD, Expert CAD, Knowledge-based CAD, etc. has continued and evolved during the years; however, the early ICAD system and the other KBE platforms that have followed, possibly represent, still to date, the most successful industrial implementation of the entire “intelligent CAD” concept.

1 According to the ICAD developers, the name ICAD was not the acronym of intelligent CAD.


5. KBE systems (dis)similarities with conventional expert systems. The programming approach

In order to practice knowledge based engineering, specific software tools, called KBE systems or KBE platforms, are normally required (see KBE definition in section 2). A KBE developer uses a KBE system to build so called KBE applications: dedicated programs to solve specific problems, typically, but not necessarily, related to modeling and configuring hardware products, both in terms of geometry and metadata. ICAD, GDL by Genworks, AML by Techno-soft and Knowledge Fusion by Siemens NX are just some examples of commercial KBE systems. Since the market of KBE software is very dynamic, the reader can refer to [39] for up to date information on current availability.

A KBE system, similar to an expert system shell (see section 3), is a general purpose tool, hence does not contain any knowledge about any specific domain (apart from the knowledge required to generate and manipulate a certain amount of primitive entities, including geometric objects such as points, boxes, cylinders, etc.). Whilst an expert system shell allows the user to fill the knowledge base via user interface, while limiting or eliminating the need of authoring and editing a full-text representation of the knowledge, typical KBE systems have a different approach, in which a full structured-text format and semantics, together with highly refined and specialized text-editing tools are provided. This approach has potential to empower the user generating the most flexible, detailed and dynamic domain knowledge models, as required to support the design of even very complex products. The typical use of KBE technology to support engineering design will be discussed in Section 7, where a few examples of KBE applications developed for aerospace design are provided.

Developing a KBE application is mostly about writing code using a KBE programming language. State-of-the-art KBE systems provide the user with an object oriented language, which allows modeling the domain knowledge as a coherent network of classes. Details on the characteristics of KBE languages will provided in Section 6. Screenshots of the development interface of GDL are given in Fig. 4, as example of the typical working environment for KBE applications development. Generally the development interface is composed of one customized text editor to write code and one or more graphical browsers to test, query and visualize the results produced by the given KBE application. The actual KBE development system runs behind the scenes and includes a language compiler, a debugger and a runtime environment.

Although there are similarities between the development of a KBE application and software development by means of a general purpose programming language, there are also fundamental differences. One of the most outstanding features of KBE systems is that their programming language can be used to drive a CAD engine, which is either integrated or tightly connected to the KBE system itself. This capability to generate and manipulate geometry represents also the most evident distinction of a KBE system with respect to conventional KBS. Details on the use of KBE languages to define geometry generative rules will be provided in Section 7. Table 1 gives an overview of the geometry modeling kernels (i.e., the CAD engines) integrated or connected to the four KBE systems mentioned above.

Once a KBE application has been finalized, it can be built into a so called runtime application and deployed as a conventional computer aided engineering (CAE) tool, possibly with a dedicated user interface. In this form, designers, engineers and others involved in
the engineering design process can simply use it, without being confronted with the syntax of the programming language running “under the hood”, neither requiring a KBE development license to be installed on their machine.

Differently than conventional rule based systems, but likewise frame based systems, a KBE application shows no crisp separation between knowledge and inference mechanism. The domain knowledge and the control strategy to access and manipulate it are strongly intertwined. Again, differently from RBSs, but rather similarly to FBSs, KBE systems leave the freedom – but also the burden – of modeling the knowledge domain to the developer. That is, the selection of the adequate levels of abstraction and the definition of the proper networks of classes and objects is fully in the hands of the engineer. It follows that expanding, updating and maintaining a KBE application is not just adding or deleting rules from a list. Although the OO approach is claimed able to provide “a sustainable way of writing spaghetti code [40]”, it is the competence of KBE developers writing applications that are clear
and well structured, hence scalable and maintainable. Eventually, the high level of flexibility and control provided by the OO approach is exactly what is required to build applications that are fully tailored to the specific needs of the user and the peculiarities of their domain of interest.

### 6. Main characteristics of KBE languages

State-of-the-art KBE systems generally provide user with a programming language that supports the object oriented paradigm. As it happens, KBE languages are very often based on object oriented dialects of the Lisp programming language:

- **IDL**, the ICAD Design Language, is based on a pre-ANSI Common Lisp version, with Flavors object system (an early object oriented extension to Lisp, developed at the MIT Artificial Intelligence Laboratory [41]).
- **GDL**, the General-purpose Declarative Language of Genworks, is based on the ANSI standard version of Common Lisp and makes use of its CLOS (Common Lisp Object System) [42].
- **AML**, the Adaptive Modeling Language of Technosoft, was originally written in Common Lisp, and subsequently recoded in a proprietary—yet—LISP-similar language.
- **Intent!**, the KBE proprietary language developed by Heide Corporation and now integrated in Knowledge Fusion, the KBE package of Siemens NX (formerly Unigraphics), belongs also to the family of Lisp-inspired languages.

This non accidental occurrence of Lisp in the KBE area is just another strong clue of the AI roots of knowledge based engineering. Although Lisp is the second oldest programming language after FORTRAN, to date, it is still the favored programming language for artificial intelligence research and implementation [40].

The name Lisp derives from **LISP Processing**; indeed, lists are one of Lisp languages’ major data structures, and Lisp source code is itself made up of lists. As a result, Lisp programs can manipulate source code as a data structure, giving rise to the **macroexpansions** systems that allow programmers to create new syntax or even new “little languages” embedded in Lisp [43]. Although Lisp is by itself a high-level language, it can be used to build even higher-level layers of language on top of itself. The results are called embedded languages and are typically Lisp supersets, of which KBE languages like IDL and GDL just represent two outstanding examples. The advantage of using a superset is that, while the full Lisp language is always available, new special operators are available to provide the user with even higher level and user-friendly language constructs.

Actually, the availability of these operators (and associated runtime machinery) represents the core of the added value of a KBE language with respect to raw Lisp. An example of the most important operator is illustrated in detail in section 6.1. Sections 6.2—6.3 discuss other relevant characteristics of KBE languages, which are partially derived from Common Lisp and the object oriented paradigm in general. Sections 6.4—6.6 highlight the added value of a KBE language and illustrate the powerful and distinctive features that make it much more suitable to support engineering design than a conventional general purpose programming language.

### 6.1. KBE operators to define classes and objects hierarchies

Possibly the most relevant operator provided by state-of-the-art KBE systems is the one used to define classes and objects hierarchies.). Mastering the use of this operator, which belongs to the category of so called “definers”, is fundamental for the development of any KBE application, in almost any of the KBE systems on the market. As a representative case, the GDL operator called **define-object** is discussed here. Indeed, ICAD, AML and Knowledge Fusion (KF) provide very similar versions of the same definer, although different names and syntax are used. This outstanding language similarity is highlighted in Table 2, where the structure of the ICAD **define-object**, the Knowledge Fusion **defclass** and the AML **define-class** are shown as well.

The **define-object** operator (as well as its aforementioned counterparts) 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. The code of a pseudo-KBE application (developed in ICAD) is provided in Appendix A, where, as example, the **define-object** operator is used to define the top level architecture of an aircraft and compute some performance characteristics.

The basic syntax is the following: **(defineobject name (mixins) specifications)**, where:

---

**Table 2**

<table>
<thead>
<tr>
<th>IDL</th>
<th>Knowledge Fusion</th>
<th>AML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input-slots</td>
<td>Input</td>
<td>defclass</td>
</tr>
<tr>
<td>Input-slot:etable</td>
<td>Default-inputs</td>
<td>Any data type followed by the behavioral flag parameter (plus optional default value)</td>
</tr>
<tr>
<td>computed-slots</td>
<td>attributes</td>
<td>Specification of several data types, plus an optional behavioral flag (e.g., lookup, uncached and parameter)</td>
</tr>
<tr>
<td>computed-slot:etable</td>
<td>Modifiable-attributes</td>
<td>A data type followed by the behavioral flag modifiable</td>
</tr>
<tr>
<td>Trickle-down-slots</td>
<td>Descendant-attributes</td>
<td>All attributes descendant by default</td>
</tr>
<tr>
<td>Type</td>
<td>Class</td>
<td></td>
</tr>
<tr>
<td>Objects</td>
<td>Child</td>
<td></td>
</tr>
<tr>
<td>Hidden-objects</td>
<td>Sub-objects</td>
<td></td>
</tr>
</tbody>
</table>

4. Differently than ICAD and GDL, KF does not support dynamic typing (see Section 6.2). Hence the type of the attribute must always be specified, e.g., number, string and Boolean.
5. A behavioral flag might be used to specify the behaviour of an attribute. The flag parameter is used to create a correspondent of the input or input-slot keyword.
6. These objects will not be visualized in the object tree. When % is used as the first character of an attribute name, such attribute will not be visible from outside the object definition.
• **Name** is the user defined name of the class being created.

• **Mixins** is a list (it can be empty) of other classes (identified by their name), whose characteristics will be inherited by the class being defined. The mixins can either be formal superclasses, of which the class being defined is a true specialization, or other classes whose attributes and component objects will be inherited.

The specifications include the following items:

- **Input-slots**: list of property-values pairs to be assigned in order to generate specific instantiations of the class. These represent the so called class protocol, i.e., the minimum list of values to be “passed in” in order to create an instance. This list can also be empty or have default values specified.

- **Computed-slots**: list of property-values pairs where the values are expressions that return the property value once evaluated. These expressions can be production rules, mathematical formulas, engineering relations, etc. (examples will be provided in Section 7), and can refer to the return-value of other slots defined either in the same object (i.e., input and computed-slots), or in any child object (see next bullet) or in any mixins and relative objects. When slots of other objects are required, their reference chain will have to be specified (see definition below).

- **Objects**: list of child objects, i.e., the component objects that will be part of the class instantiation. For each child object, the following must be specified:
  - The child object name
  - The type of the child object, i.e., the name of the class to be instantiated in order to create such an object
  - Pairs of property names and values, where the property names will have to match the input slots of the class specified by the keyword type (i.e., they must sufficiently match the protocol of the child class) and the values are expressions, which will be evaluated when instances of the child objects are required. Similar to computed-slots, these expressions can refer to any slot defined in the given define-object expressions and its mixins, or, by means of the reference chain specification, to any slot of any child being defined or any other descendant.

- **Functions**: list of functions, each defined by a name, a list of arguments and a body. Differently than standard Lisp functions, these operate within the given define-object and, similarly to a computed-slot, their body can refer both to slots of the given define-object or to any other object slot. In this second case, the reference chain of the given slot will have to be specified (see definition below). The main differences with computed slots are that (1) functions can take arguments, and (2) the result of a function is not cached, hence is not occupying any memory. The caching mechanism is discussed in section 6.5.

Eventually, all KBE applications consist of a number of define-object-like declarations, properly interconnected as required to model even very complex products. This whole network of define-object’s, in KBE terminology, is called the product model or generative model (more details in Section 8). The hierarchical structure of objects obtained by an instantiation of the product model is called the object tree or product tree. See in Fig. 5 an example of a product tree, generated by the instantiation of a blended wing body aircraft product model developed in ICAD [44]. Another example of product tree is provided in Fig. 4 (see the tree pane in the GDL graphical browser).

The aforementioned reference chain is nothing else than the list of objects to be sequentially accessed down the object tree (from root to leaves) in order to obtain the value of a given slot. In the product tree example of Fig. 5, the root object Bwb, in order to get the value of the slot weight of the object Rib 2, which is part of the object fuselage 0, which is part of the object fuselage, etc., would need to specify the following reference chain: *(the fuselage (fuselage-structure 0) (rib 2) weight)*.

It is interesting to note that this referencing mechanism is the same used in frame based systems to access slot values across hierarchies of object frames (see example in Fig. 2). Actually, the entire operator described above shows a large similarity with the concept of frame addressed in Section 3. Looking at the details, even the GDL term slot seems a legacy of frame based systems.

6.2. Flexibility and control: dynamic typing, dynamic class instantiation and objects quantification

One of the characteristics of KBE languages based on Lisp, such as IDL and GDL, is that values have types, but attributes (i.e. slots) do not, at least not necessarily. Hence, differently than many general purpose programming languages like FORTRAN, slot types do not need to be declared ahead but can be simply established on the fly, based on the types of observed return-values. For example, the value of a given attribute can change at runtime from a Boolean like “NIL” to an integer. This programming style is commonly addressed as dynamic typing [42,45].

In a KBE language, dynamic typing assumes an even higher level of reach. Not only slots, but even an object type can change dynamically. Although child objects must have their types specified at definition time, the type specification need not to be hardcoded. In addition, it is very easy to define sequences of child objects, where also the number of instances can change, depending on the runtime evaluation of specific rules. It follows that defining sequences of objects in a KBE system is different from just creating “carbon copies” of the same part/feature, as typical in many CAD systems. KBE systems allow the instantiation of variable length sequences of objects, where each single member can be defined using different parameter values, as well as different parameter sets, given that each member can be instantiated from a different class.

The outstanding result is that the topology of the product tree, i.e., the number of levels in the product tree, as well as the type of objects in the tree, is not fixed but is reconfigurable at runtime, hence, after the model has been instantiated and without the need to restart the application. The whole KBE model is fully dynamic by nature.

6.3. Declarative coding style

When authoring a piece of code using a KBE language, in general there is no “start” or “end”, hence the order in which slots are declared and objects listed is not relevant. An attribute that defines the total weight of an assembly can be specified in the code, before the attributes defining the weight of the assembly components. The program interpreter/compiler will figure out at runtime the correct order to trigger attributes evaluation and object instantiation (see Section 7.1 for demand driven evaluation). This declarative style is opposed to the so called procedural style, used in FORTRAN, for example. Here any procedure has to be defined step-by-step using only the correct temporal order of the events.

It should be noted that the KBE languages based on Common Lisp can actually support both styles, thereby they are called multi-paradigm languages. While the advantages of declarative coding are evident both during coding and at runtime, a local switch to the procedural approach is useful in such cases as “IF Fact A is True THEN first Do this, then do that and subsequently do the other”.

- **Functions**: list of functions, each defined by a name, a list of arguments and a body. Differently than standard Lisp functions, these operate within the given define-object and, similarly to a computed-slot, their body can refer both to slots of the given define-object or to any other object slot. In this second case, the reference chain of the given slot will have to be specified (see definition below). The main differences with computed slots are that (1) functions can take arguments, and (2) the result of a function is not cached, hence is not occupying any memory. The caching mechanism is discussed in section 6.5.

Eventually, all KBE applications consist of a number of define-object-like declarations, properly interconnected as required to model even very complex products. This whole network of define-object’s, in KBE terminology, is called the product model or generative model (more details in Section 8). The hierarchical structure of objects obtained by an instantiation of the product model is called the object tree or product tree. See in Fig. 5 an example of a product tree, generated by the instantiation of a blended wing body aircraft product model developed in ICAD [44]. Another example of product tree is provided in Fig. 4 (see the tree pane in the GDL graphical browser).

The aforementioned reference chain is nothing else than the list of objects to be sequentially accessed down the object tree (from root to leaves) in order to obtain the value of a given slot. In the
6.4. The object-oriented paradigm and beyond

In order to model very complex products and manage efficiently large bodies of knowledge, KBE systems largely tap the potential of the object oriented paradigm of their underlying language. Indeed, some of the powerful features of KBE systems are often erroneously assumed to be a KBE prerogative, whilst they are just a legacy of the object oriented paradigm.

The abovementioned features of dynamic typing, declarative coding and multi-paradigm, for example, are contributed directly by the OO facility of Common Lisp.

The possibility to use mixins for the specification of many superclasses is because KBE languages like IDL, GDL, AML, etc. support the OO multiple inheritance concept.

The expressions used to evaluate slots, correspond to OO methods and computed slots represent one mechanism of what in OO parlance is called message-passing behavior. Objects interact by sending messages to each other, and each object attribute represents a message the given object is able to answer [46]. In other words, sending a message is nothing more than sending an object a specific request to evaluate one of its attributes (slot). The reference chain addressed in section 6.1, represents the address of the object where to send a message.

A clarifying note on the concepts of child objects and inheritance might be necessary to avoid a common misunderstanding. In KBE parlance, children (i.e. child objects) do not inherit from their parents. Inheritance is an exclusive relation between a class and its superclasses. Only parameters can be passed down from parents to children, which is the main mechanism to have information flowing down the object tree. In synthesis, by inheritance, parameters are transmitted across hierarchies of classes (i.e., hierarchies linked by the relationship is-a); by passing down, parameter values flow down part-whole hierarchies of objects (i.e., hierarchies linked by the relationship has-part).

On the basis of the discussion above, it might seem that most of the merits claimed by KBE systems actually belongs to the OO nature of their underlying language. Thus, what is the added value of a Lisp superset like GDL, for example, with its define-object operator?

First, it is the ability to provide a much simpler, user friendly and intuitive way of authoring complex hierarchies of objects, without requiring engineers to possess the programming skills of a Lisp expert. It should not be forgotten that engineers are the target users of KBE systems! In this respect, an operator like Define-object delivers a high level interface to the Common Lisp object facility and maps any GDL defined object to an actual Common Lisp object. Actually, behind the concise definition created by such a “definer” operator, there is hidden a voluminous and opaque chunk of Lisp code, which, through the macroexpansion mechanism, automatically expands at compile time, (luckily) in a way that is fully transparent to the user. Some macroexpansion examples are given in [46].

Second, operators like the GDL define-object bring some features that are not present in raw Common Lisp, or any other standard OO language, but represent an exclusive characteristic of KBE technology. One of these features concerns the ability to generate and manipulate geometry and is discussed in section 0. Another deals

---

**Fig. 5.** Example of product tree generated via instantiation of a blended wing body aircraft product model (only few branches fully expanded).
with a powerful mechanism for efficient data and information management and is addressed in the next section.

6.5. KBE specific features: runtime caching and dependency tracking

An operator such as define-object (or its direct counterpart in other KBE languages) not only provides a high-level interface to the underlying language objects facility, but also brings in the so-called runtime caching and dependency tracking mechanism, which is not natively present even in a powerful OO language such as Common Lisp, but represents a distinctive feature of true KBE languages.

Caching refers to the ability of the KBE system to memorize at runtime the results of computed values (e.g., computed slots and instantiated objects), so that they can be reused when required, without the need to re-compute them again and again, unless necessary. The dependency tracking mechanism serves to keep track of the current validity of the cached values. As soon as these values are no longer valid (stale), they are set to unbound and recomputed if and only at the very moment they are again demanded.

This dependency tracking mechanism is at the base of associative modeling, which is of extreme interest for engineering design applications. For example, the shape of a wing rib can be defined accordingly to the shape of the wing aerodynamic surface. In case the latter is modified, the dependency tracking mechanism will notify the system that the given rib instance is no longer valid and will be eliminated from the product tree, together with all the information (objects and attributes) depending on it. The new rib object, including its attributes and the rest of the affected information, will not be re-instantiated/updated/re-evaluated automatically, but only when and if needed (see demand driven instantiation in the next section).

In conventional programming, these mechanisms would need to be re-coded by the application developer, which is a non-trivial programming task, neither it would guarantee the same level of speed performance [47]. A true KBE language does it automatically and transparently to the user.

Whilst in the early age of KBE, the caching mechanism was often the cause of a too large memory consumption, the evolution of commodity computers has eliminated such issue. In addition, every time a value or an object becomes stale, the Lisp garbage collector (in case of Lisp based KBE languages) takes care of claiming back the relative space in memory. Also, this happens automatically and transparently to the user, who is spared from any memory management duty.

6.6. KBE specific features: demand driven evaluation

In general a KBE system has two possible ways of operating, namely by eager or lazy evaluation, which can be compared to the forward and backward chaining inference mechanisms of classical knowledge based systems [22].

By default, KBE systems use the lazy approach, better known in the field as demand-driven approach. That is, they evaluate just those chains of expressions required to satisfy a direct request of the user (i.e., the evaluation of certain attribute/slot or the instantiation of an object), or the indirect requests of another object, which is trying to satisfy a user demand. For example, the system will create an instance of the rib object only when the weight of the abovementioned wing rib is required. The reference wing surface will be generated only when the generation of the rib object is required, and so on, until all the information required to respond to the user request will be made available.

It should be recognized that a typical object tree can be structured in hundreds of branches and include thousands of attributes. Hence, the ability to evaluate specific attributes and product model branches at demand, without the need to evaluate the whole model from its root, prevents waste of computational resources [1, 2, 46]. The demand driven approach is not only beneficial at runtime, but it also supports code prototyping. Indeed, it allows the developer to focus on a limited part of the KBE application, while the rest of it can be left incomplete or even incorrect. Since this latter part will not be evaluated automatically at run time (unless explicitly demanded), it will not generate any error preventing the developer from testing the sole branches of interest.

As mentioned above, KBE systems can also “eagerly” compute all the chains of values affected by any occurred change to a given attribute or object. In this case it will work similar to a spreadsheet application where, if a cell value changes, the values of all the linked cells are automatically updated. This capability is particularly convenient to define so-called event triggers or demons, as they are generally referred to in the field of frame based systems [22]. These are silent processes that automatically activate in case of certain events. For instance, a message can be visualized in case a constraint or a rule has been violated, or a certain procedure is started when the user clicks a certain area of the screen.

7. The rules of knowledge based engineering

In KBE parlance, all the possible expressions used to define attributes (slots), specify the number and type of objects, communicate with external tools, and so on, are addressed with the generic term of rules (or engineering rules). For this reason, KBE is often addressed as a technology to perform rule based design. There is nothing wrong with this description of KBE, so long as the fundamental difference with conventional rule based systems is acknowledged, where all rules are of the type IF-THEN and there is a crisp separation between reasoning mechanism and knowledge base.

Within the large and heterogeneous set of KBE rules, the following main typologies can be identified (although the following examples are GDL specific, equivalent expressions are available in most of the existing KBE languages):

Logic rules (or conditional expressions): Apart from the basic IF-THEN rule (production rules), KBE languages such as GDL, AML, and IDL provide more sophisticated conditional expressions, often inherited from Common Lisp. Two examples are given in Table 3.

Math rules: Any kind of mathematical rule is included in this group, including trigonometric functions and operators for matrices and vector algebra. Basic mathematical operators such as ±, ×, ÷, or ^, are just Common Lisp functions. Many others are functions and operators provided by the given KBE language. In the example of Table 3, it can be noted the use of the prefix notation and the absence of the symbol “=”, which is a specific Common Lisp function to check whether two numbers are the same. In the example below, “<2” is a KBE language specific operator (not a native Lisp operator). These rules are commonly used for evaluating computed-slots and child object inputs. Of course, mathematical rules can be used both in the antecedent and consequent part of any production rules.

Geometry manipulation rules: In this category rules for generation and manipulation of geometric entities are included (examples in Table 4). These are language constructs for the generation of many and diverse geometrical entities, ranging from basic primitives (points, curves, cylinders, etc.) to complex surfaces and solid bodies. It is important to note that all instantiated geometry entities are real objects, as such they can answer messages like length, area, volume, center of mass, etc. Also each point, curve, face, etc. which is a component of a given entity can answer messages like parametric or Cartesian coordinates, radius of curvature, control points, normal vector, etc. Manipulation rules allow performance of all kind of operations on any defined geometric entity, such as
projections, intersections, extrusion, trimming, merging, etc., as examples of logic and math rules.

Table 3

<table>
<thead>
<tr>
<th>Example rule</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>case expression (test consequent) &amp;optional otherwise otherwise-expression</td>
<td>Logic rule that returns consequent if expression evaluates to test. Any number of tests can be defined. If expression does not evaluate to any test, it returns the otherwise-expression (if supplied) or nil.</td>
</tr>
<tr>
<td>cond [(test consequent) ...]</td>
<td>Logic rule that returns consequent if test evaluates to true. If no test evaluates to true, this returns nil.</td>
</tr>
</tbody>
</table>

The mathematical expression:

\[ L = \frac{1}{2} \sqrt{L^2 - L_1^2} \cdot C_1 \]

Table 4

<table>
<thead>
<tr>
<th>Example rule</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Define-object container (box) computed-slots/length 10 width 20 height (* (the length) (the width)))</td>
<td>Definition of a class called container, which is of type box, a predefined geometry primitive. Height defined by means of a parametric rule (i.e., an expression).</td>
</tr>
<tr>
<td>(Define-object wing (lofted-surface) curves (list (the root-airfoil) (the intermediate-airfoil) (the tip-airfoil)))</td>
<td>Definition of a child object called my-curve, as the intersection curve between two previously defined surfaces, first-surface and second-surface.</td>
</tr>
<tr>
<td>:objects (my-curve :Type 'surface-intersection-curve :surface-1 (the first-surface) :surface-2 (the second-surface))</td>
<td>Definition of a class object called my-curve, as the intersection curve between two previously defined surfaces, first-surface and second-surface.</td>
</tr>
<tr>
<td>:computed-slots (distance-object (the wing-surface (minimum-distance-to-curve (the reference-curve))))</td>
<td>Definition of a class attribute called distance-object, as the minimum distance between previously defined surface and curve.</td>
</tr>
</tbody>
</table>

Table 5

<table>
<thead>
<tr>
<th>Example rule</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>:objects (my-beam :type 'beam :sequence (:size (round (div (the total-load) (the allowable-beam-load))) :beam-length (IF (eql (the-child index) 1) 10 8)))</td>
<td>The number of beam instantiations is computed based on the evaluation of a mathematical rule. The attribute beam-length of each object in the series is evaluated, based on a production rule.</td>
</tr>
<tr>
<td>:objects ((my-beam :type 'beam :sequence (:size (IF (&gt; (the total-load) 100)) 3 2))</td>
<td>The number of beams is computed based on the evaluation of a production rule. A mathematical expression is computed to evaluate the premises.</td>
</tr>
<tr>
<td>:objects (aircraft-tail :type (IF (the tailies?) 'nil 'conventional-tail))</td>
<td>The class of the object tail is selected at runtime based on a logic rule. The object is not generated if the Boolean attribute tailies? is not true.</td>
</tr>
</tbody>
</table>

For example, as anticipated in section 6.1, the so-called KBE product model represents the core and essence of any KBE application. It consists of a structured and dynamic network of classes, where both product and process knowledge, both geometry-related and non-geometry related, are modeled using a broad typology of rules.

A product model is a generic representation of the type of product for which the KBE application has been created, thereby it is kinds of files to parse and retrieve data and information to be processed within the KBE application. Other rules exist to create files containing data and information generated by the KBE application. For example, it is possible for a KBE application to generate as output standard geometry exchange data files like STL, IGES and STEP; or XML files or any sort of free format ASCII files for more textual data. Rules also exist to start at runtime external applications, wait for results, collect them and return to the main thread.

The particular significance and enormous potential offered by these geometry manipulation and configuration selection rules need to be acknowledged:

1. Geometry handling rules do not generally exist in any classical knowledge based systems, because their evaluation requires the notions of space and relative positioning of assembly/parts/features. As discussed in section 0, this new capability provides an essential condition to the exploitation of knowledge based systems in the field of engineering design.

2. Geometry manipulation rules allow performing and recording by means of a concise programming language, operations that in classical parametric CAD systems require intensive sessions of mouse clicking and menu items selections.

3. The possibility to combine geometry handling rules with logic and configuration selection rules adds a new dimension to the controllability and flexibility of any defined product model. Spatial integration issues can be systematically checked and corrected/prevented at runtime to enable significant variations of the product configuration. These rules are generally not available even in the most advanced CAD systems and represent an extremely powerful prerogative of KBE.

The availability of a programming language to express, manipulate and combine the sorts of rules mentioned above is the key to enable the generative design approach illustrated in the next section.

8. KBE product models for generative design

As anticipated in section 6.1, the so-called KBE product model represents the core and essence of any KBE application. It consists of a structured and dynamic network of classes, where both product and process knowledge, both geometry-related and non-geometry related, are modeled using a broad typology of rules.

A product model is a generic representation of the type of product for which the KBE application has been created, thereby it is
also addressed as metamodel. It is not made up of fixed geometric entities, with fixed dimensions, in a fixed configuration, but it contains the engineering rules that at runtime will determine the design of the product [1].

The functionality of the product model is illustrated by the simplified representation of Fig. 6 from [1]: a set of input values is assigned to the parameters that are used in the product model, the KBE system applies the rules to process the input values and, finally, the engineered design is generated, with little or no human intervention. This is typically addressed as generative design, where the product model is also known as generative model.

One of the significant advantages of the generative approach is that the set of input data has a univocal correspondence with the generated output: any time the product model is instantiated with the same set of input values, the same rules will be systematically evaluated, the same objects will be generated, and, finally, the same results will be obtained. Conformity to all the rules recorded in the product model is always guaranteed, and engineers can be confident that each resulting design is the outcome of a thoroughly formalized and deterministic approach. The product model provides the clear reason for every dimension, design decision and configuration feature contained in the final output. This represents invaluable information, both for the designer and for those who review the design.

The example of Fig. 6 suggests the capability of a single KBE application to deliver automatically a fully engineered product, complete with relative digital documentation, just from a list of input parameters. This optimistic vision was the selling point of the early KBE vendors and the goal pursued by the first KBE practitioners. Although success stories of fully integrated KBE design tools can be found in the literature [1], a different approach is generally required when dealing with complex products requiring complex multidisciplinary analysis and optimization [21,48]. In these cases, modular, heterogeneous and distributed design frameworks have much higher chances of success, where KBE is used only in those areas where it has a clear competitive advantage, that is, for repetitive, rule based design cases of multidisciplinary nature, requiring geometry manipulation and product (re)configuration. To be noted is that not all these characteristics need to be present simultaneously, as far as at least one is dominant.

To this purpose, Ref. [4,49–51] offer examples of distributed design frameworks to support aircraft multidisciplinary design optimization (MDO), where KBE has been mainly used for the development of so-called multi model generators (MMGs). These are KBE applications able to generate automatically models of a specific family of products, e.g., aircraft movable components (Fig. 7 from [51]) or complete aircraft configurations (Fig. 8 from [21]), and, for each model, to create the discipline abstractions required by the various analysis tools in the framework. Most of these analysis tools, as well as the majority of the preliminary sizing modules and the optimizer, generally, are not KBE applications, but commercial off the shelf tools (e.g., NASTRAN) and other in-house developed apps (e.g., based on MATLAB).

Fig. 9 illustrates the role of the MMG developed within the European project MOB [52], to support distributed multidisciplinary design optimization of blended wing body aircraft configurations. In that occasion, the MMG was developed using the ICAD KBE system. That KBE application was able to model topological and morphological variants of blended wing body aircraft, and to automate the generation of a multitude of diverse, yet coherent, sub-models, as required to feed the suite of analysis tools operated by the project partners. Based on knowledge previously acquired by experts, the MMG enabled the automation of those lengthy and repetitive preprocessing activities required to generate models for both low and high fidelity aerodynamic models, 2-D planform models for aeroelastic analysis, and structural models for finite element analysis. The MMG was able also to automate the definition of the design variable groups for structural optimization, extract models of structural details (e.g., door cut outs) for multi-level optimization and compute the mass distribution of systems and fuel tanks for the weight and balance assessment of the aircraft. Once the MOB computational framework was in place, more than 50 aircraft variants could be evaluated, by means of both low and high fidelity analysis tools, totally hands off, in the time frame of just

![Fig. 6. The product (or generative) model of a KBE application takes input specifications, applies relevant rules and produces automatically an engineered design.](image-url)
a couple of days. Without the enabling role of the KBE application, such a MDO exercise would have taken months.

Whatever the complexity of the design framework wherein the given KBE module operates, or of the product model itself, the manner in which KBE enables generative design does not change substantially. The user must always start creating an instance of the class at the root of the product tree. To do that, values will have been assigned to the input attributes of that class, generally via some kind of input file(s) or, interactively, by means of a user interface. Once the root object has been instantiated, there are two main options:

1. **Forcibly** compute all or some of the branches of the product tree (in a progressive manner, from root to leaves). The attribute values required for the instantiation of each (sub)branch are automatically evaluated and systematically passed down from parent to child.

2. Request the product model to return the value of a certain attribute or to generate some output file among the list of outputs that the KBE application has been developed to accommodate (such as the bill of materials in the example of Fig. 6). In this case, the demand driven mechanism automatically triggers the evaluation of the sole product tree objects and attributes.

---

**Fig. 8.** Examples of different aircraft configurations and variants, all generated as different instantiations of the same product model, i.e., the Aircraft multi model generator.

**Fig. 9.** Role of the MMG within the MOB framework for multidisciplinary design optimization of blended wing body aircraft configurations. The MMG generates on demand dedicated models for a large set of distributed analysis tools, both low and high fidelity, in-house developed and commercial of the shelf.
that are necessary to satisfy the request. For obvious reasons of computational efficiency, this latter approach is more common in practice.

State-of-the-art KBE systems allow the execution of the whole process (product model instantiation, request submission, execution and output generation) also in batch mode, i.e., completely hands-off, which is actually the essence of generative design. Sending batch run requests to the product model can either be done by a human, or another software tool. The latter case is typical (and necessary) when the given KBE application operates inside an optimization framework. Fig. 10 shows the operational approach of the aircraft MMG addressed in Figs. 8 and 9. Both the interactive and batch mode control are indicated. In the second case, the user will have to indicate via input file, the list of required outputs among those offered (see the file on the right, in the input files folder).

Latest generation KBE systems allow delivering applications as web services on remote servers, according to the Software as a Service paradigm. In this case the KBE application is constantly available on the server and can accept HTTP requests. One example is given in [49].

9. On the opportunities of the programming approach and the long lasting KBE vs. CAD debate

As evident from the previous sections, practicing knowledge based engineering (not using KBE applications!) is mostly about programming software applications to enhance the level of automation in the engineering design process.

In the early years of KBE, the presence of this new breed of tools for design was creating quite some “irritation” in the CAD development area. In particular, the KBE ability to manipulate geometry was possibly experienced as an attack on the hegemony of CAD systems in the field of engineering design. As Cooper indicates in Ref. [53], the original KBE systems were indeed created in response to a lack of capability in CAD systems and, because of this, their marketing and overall positioning tended to be CAD-oriented. Unfortunately, these CAD-like capabilities have often been misused to create the misconception that KBE systems are “just CAD systems” - of a very inconvenient species - where users are forced to write in rules with syntax what could be done more easily and intuitively with the fancy GUI of a true CAD system. For a long time, the imposed use of the programming approach was used as a major item into the discredit campaign of CAD vendors against KBE.

It is interesting to observe that, eventually, also CAD developers have acknowledged the use of a language to be the only way to fulfill certain advanced and specific customer needs. In order to provide users with some design automation capabilities, yet without spoiling the simplicity of use of the tool, most advanced CAD systems, nowadays, offer the possibility to record interactive sessions (click by click), or, even more, to write programmable macros and make use of function calls to external routines (e.g., written in Fortran, C, C++ and Visual Basic). However, the results are seldom at the level of true KBE applications, where the programming approach is native. For example, Visual Basic macros are orders of magnitude slower than true KBE applications, because they are interpreted and not compiled [53]. Furthermore, these macros have a limited reach. The only way to access and manipulate all the features of the CAD system typically requires hardcore programming at API level. By using powerful languages such as C++ fast and efficient pieces of CAD automation code can be written. However, the programming skills required to perform such CAD systems customizations are often higher than those required to develop standard KBE applications, which contradicts the initial claim of CAD use simplicity.
Although, CAD and KBE do share some capabilities, they have a fundamentally different focus: CAD systems focus on geometry. KBE systems focus on rule capture and knowledge, with geometry merely being one of the many kinds of inputs that can be passed in and outputs that can be generated [31]. There are estimates that less than 50% in a KBE application is directly related to geometry [2]. It follows that a discussion on the convenience of the KBE programming approach against the interactive operation of a CAD system is fundamentally misdirected. The point is about acknowledging when one approach is more convenient than the other and how to exploit their synergy:

- When the focus is just on geometry generation and manipulation; when aspects such as direct interaction with the geometry model, visual rendering and inspection are essential; when it is about one off design and aesthetics and heuristics are driving the modeling effort, rather than engineering rules; when model variants can be controlled just using simple mathematical expressions and basic parametric relations; when the focus is on detailed design, which can benefit from the use of feature and component plug-in libraries, then, the use of conventional CAD is more convenient that deploying a KBE system.

- When it is about capturing design intent rather than a design output, a language is required. To bake a cake, the picture of a baked cake is not sufficient, the recipe is necessary. In this case the programming approach of KBE systems offers the best solution. The actual capability of expressing design intent using a language is demonstrated, for example, by the configuration selection rules described in section 0. Whilst, CAD systems aim at providing the best recording of the final result of a human design process, KBE systems aim at recording the human design process (i.e., the design intent) and not just its end result.

- When it is about supporting automation while guaranteeing consistency, a language is required. Having a KBE generative model is like having a process recorded on some kind of playable medium. Every time the generative model is “played”, it is guaranteed that the same process is consistently repeated (i.e., the same rules and reasoning mechanisms are applied) for different valid input values, whoever the operator and however large the number of re-plays. There are many cases in engineering design, such as design optimization, where having the human in the loop is only an obstacle to automation and a potential source of errors.

- When it is about integration flexibility with external design and simulation tools, a language can offer a competitive advantage. Both CAD and KBE systems are generally able to link (to each other and to other tools via standard data exchange format such as IGES, STEP, etc. However, as soon as ad hoc, ASCII based exchange files are required, a full-featured programming language represents the natural solution to generate dedicated writers and parsers.

- When the design of a given product features aspects of aesthetics and requires the generation of details, but its configuration and sizing require a multidisciplinary analysis and optimization approach, combinations of CAD and KBE applications can offer the best solution. In this case the geometry drafted with a CAD system can become the input for a KBE application, which takes care of supporting the complex multidisciplinary analysis (and optimization) process and feedbacks the engineered product to the CAD system. Here, the final, interactive detailing work can take place.

In the fields of aerospace and automotive, there is a wealth of opportunities to deploy KBE for improving the level of automation in design by capturing complex, non-creative and repetitive engineering processes. However, the use of one selected high-end CAD systems to record, share and communicate product geometry remains compulsory. Furthermore, the use of such selected CAD system is made mandatory to all suppliers as well. In this case, there is no discussion of choosing between CAD or KBE.

In the final analysis, interaction and automation, heuristic and by-the-rules, geometry related or not, one-off and repetitive are coexistent and interrelated aspects of the design process: CAD and KBE can both contribute to this process and their smooth integration must be a point of attention to both CAD and KBE system developers.

### 10. Evolution and trends of KBE technology

As KBE systems grew and became full general-purpose programming environments, clearly they fell into a category different from CAD systems. Unfortunately, the association in the marketplace with “plain old” CAD has persisted for many years and severely limited the market penetration of KBE.

The first commercial KBE system arrived on the market in the 1980s; however, KBE technology has only started to be used seriously during the last 10–15 years. Notwithstanding its huge potential, KBE was not able to score the same market success of CAD systems in their first 15 years. The reason for this limited KBE technology success can be attributed to a combination of the following causes:

- **High costs of software licenses and needed hardware:** The cost of a full development license of a first generation KBE systems was around $100000 per year. The same amount was necessary to acquire the machines to run such systems (Lisp Machines). Only in the early/mid 1990s Unix workstations came on the market for a cost one order of magnitude lower.

- **General lack of literature, study cases and metric:** The limited amount of KBE dedicated literature, the difficulty to find useful information about KBE success stories, as well as the lack of a good metric [54] to estimate the possible advantages of KBE compared to traditional methods, have for many years restrained management from taking a leap of faith.

- **Lack of KBE development methodology:** Many of the first KBE applications were written ad hoc, hence restricting their success to the very short term. No defined guidelines, no standard procedures and techniques were available to help developers generating effective applications, reusable and maintainable [16,21]. Interestingly enough, many KBE applications, whose main goal was to exploit knowledge reuse, were seldom reusable, even across similar company programs.

- **Low accessibility level:** Due to the inherent complexity of the technology (at least when compared to that of contemporary CAD systems), a different category of users and dedicated training programs were required. Indeed, the use of the programming approach demands higher abstraction ability and stronger analytical background, more typical of software developers and engineers than typical CAD operators.

- **Arguable marketing approach by KBE vendors:** As Cooper elaborates in [53], the business model of the first KBE-vendors was itself an impediment to the diffusion of their own products. Due to the large commitment required to customers to adopt such complex and expensive technology, the KBE vendors felt the need to play multiple roles. They were at the same time KBE systems suppliers, consultancy providers (to help a company to implement and operate effectively the KBE system they just sold them), as well as sellers of KBE end-user applications. As result, a third party company who wanted to use the KBE tool to offer consultancy service to some large KBE user, was becoming at the same time customer and competitor of its own KBE system vendor.
Considering the circumstances, the first successful implementations of KBE appear to be the “lucky” exploits of an elite of wealthy and broadminded organizations. However, during the years, a number of technical developments and strategy changes have created the situation for a sustainable growth of KBE within the world of industry and research:

- The cost of hardware has decreased dramatically: what yesterday required an extremely expensive Lisp Machine first and a workstation then, today runs on any laptop, at a speed tenfold higher. Many KBE vendors have specifically adapted their system to the most widespread operating systems and computer architecture, i.e., simple desktop computers running Windows. Still, only few current KBE systems can work with Linux and Mac.
- The growing complexity of the products to be engineered has nurtured the development of continuously more complex computer aided engineering tools. This trend has played in favor of KBE, because the relative complexity gap with KBE tools has automatically decreased.
- Dedicated methodologies to support a structured development of KBE applications have become available, with MOKA (Methodology and tools Oriented to KBE Applications) [14,55] as the most outstanding results in this direction. Development time reduction of 20–25% were demonstrated [20].
- Last but not least, leader companies in the development of PLM solutions have finally recognized the value of the KBE approach and augmented their top-end CAD products with KBE capabilities:
  - In 1999, PTC introduced the Behavioral Modelling toolkit for Pro/ENGINEER 2000i, which allows methods to capture rules to steer the CAD engine.
  - In 2001, UGS acquired the KBE language Intent! from Heide Corporation and embedded it into Unigraphics to form Knowledge Fusion (In 2007 UGS has been bought by Siemens PLM software).
  - In 2002, Dassault Systemes acquired KTI and their product ICAD. DS sinks ICAD and exploits KTI expertise to develop the KBE add-on’s (KnowledgeWare) of CATIA V.
  - In 2005 Autodesk acquired Engineering Intent Corporation and integrated their KBE system with Autodesk Inventor, to form Autodesk Intent (now Inventor Automation Professional).
  - In 2007, after the acquisition of the company Design Power, Bentley integrates their KBE system Design++ with Microstation.

The major branches of the KBE evolution/vendor genealogy are illustrated in Fig. 11, based on [56]. Note how, apart from GDL and AML, which are possibly the only true KBE systems on the market, all the others are basically KBE-augmented CAD systems, where a true KBE language (e.g., Siemens NX Knowledge Fusion) or some sort of KBE capabilities (e.g., Dassault Systemes CATIA V5 KnowledgeWare) have been integrated to augment the core CAD functionality. The major differences between these augmented CAD systems and true KBE systems is that the first are CAD centric, i.e., a CAD engine is always present and running, and the automation focus is largely geared towards geometry manipulation [46,57]. Hence, no data processing can occur or any kind of algorithm can be implemented that is not related or directly embedded in the definition of some geometrical object. True KBE systems, on the contrary, focus on more holistic knowledge and information manipulation, and they do not even include a real CAD kernel in their basic package, although one can be acquired under an extra license (see Table 1).

In spite of the not always orthodox KBE nature of many of the new tools on the market and the questionable absorption processes carried by the big PLM companies, there are some incontestably positive consequences on the diffusion of KBE:

- Finally, KBE is entering into the mainstream of the traditional CAD/CAE software systems, although the “KBE elements” now present in the new abovementioned systems still represent quite an exotic (sometimes esoteric) and unfamiliar aspect to most of the typical CAD users.

![Fig. 11. The KBE technology evolution and vendor genealogy.](image-url)
• The technical knowledge is now complemented by solid organization, extensive support and pervasive marketing capabilities. This is contributing to the dissemination of KBE technology to an unexpected large amount of customers, ranging from large integrators to small and medium enterprises.
• The entry cost of KBE, including hardware, licenses and training can decrease at a level that any company will have the possibility to evaluate the impact of adding a KBE system to their business.

11. Conclusions, recommendations & expectations

This paper has provided an extensive review of knowledge-based engineering, with the main purpose of increasing the level of understanding of its technology fundamentals. The AI roots of KBE have been discussed to highlight the commonalities as well as the added value of KBE with respect to conventional knowledge-based systems, in the context of engineering design. The programming approach of KBE systems and the salient characteristics of KBE languages have also been discussed in detail. A distinction has been made between those characteristics directly inherited from the object-oriented paradigm and those representing the distinctive features of KBE, such as caching and dependency tracking. A categorization of KBE rules has been provided, with particular emphasis on the object manipulation and configuration rules, which are the actual enablers of KBE. Examples have been provided to give the reader the feeling of what developing a KBE application is actually about. The convenience of the KBE programming approach to increase the level of automation in the engineering design process, to capture and record the design intent, and to facilitate design tools integration has been addressed. The similarities and main differences between KBE and CAD systems have been discussed and their specific position in the engineering design context has been indicated, as well as the opportunity to exploit these two technologies in a complementary and integrated approach. The evolution and trends of KBE technology have been illustrated, pinpointing the issues that have hampered its diffusion, notwithstanding the great potential to enhance the level of automation in the engineering design process. The recent trend of PLM developers to augment their CAD systems with KBE capabilities has been discussed and documented. The positive consequences have been highlighted, but a distinction has been made between true KBE systems and KBE-augmented CAD systems.

Looking at the actual level of capability and maturity achieved by current KBE systems, it seems that the whole concept does not need to be reinvented, but rather evolved and standardized, if even in a de facto manner. Also, new capabilities are definitely required to lower the current level of accessibility of KBE, and tools and methodologies are needed to help engineers developing better KBE applications in less time. A list of recommendations and expectations is provided below.

11.1. Consolidating the fundamental technical strengths

KBE systems developers should keep their focus on few fundamental goals:

1. Enhancing the level of robustness in the geometry manipulation process
2. Enhancing the capability to link to external CAD systems and other third party CAE tools
3. Enhancing the computational capability and stability of the system

The integration of rule-based design, geometry manipulation, and computation capability represents the real added value of KBE with respect to conventional CAD and KBSSs, as well as to other general purpose programming languages. This is the value to be preserved and strengthened.

11.2. Lowering the accessibility level

The use of a programming language to develop KBE applications must be refined but preserved, being the programming approach the most powerful and comprehensive way to control and access all the functionalities of a KBE system. On the other hand, in order to lower the accessibility level to KBE technology, the programming aspects must represent less of an obstacle for the common user. The role of KBE developer should be easily undertaken by engineers and not restricted to “natural born hackers”. To this purpose, KBE programming languages – as well accompanying development/editing environments – can play a fundamental role by continuing to automate more of what still remains too programming intensive, and attempting to get closer to natural languages. This can be achieved by increasing the amount of high level operators (such as the define-object discussed above) and predefined functions, while hiding/eliminating the use of esoteric commands and programming forms. A modernization of the accompanying development/editing environments, as well as improvements to the debugging systems of current KBE platforms would also contribute to lower the accessibility level. Furthermore, training material and documentation need to become both simpler on one hand and more complete on the other.

The purpose of lowering the accessibility level of KBE technology can and should be pursued also through the formation of better qualified engineers. While the ability to operate a CAD system is nowadays considered part of the skill portfolio of young engineers, the number of academic courses offered on KBE and similar advanced design methods is still extremely low. However, product modeling through conventional CAD or KBE systems (still) requires quite different skills, knowledge and mind-set.

11.3. Supporting interactive geometry manipulation

While KBE systems already enable drastic reduction in time to create applications, compared with plain, general purpose programming languages, this time can and should be further reduced. The potential for improvement is especially apparent in the area of geometry manipulation. Even if more than 50% of a product’s definition is often non-geometric, more than 70% of coding/debugging time is spent in geometry manipulation. The capability to have some automatic code generation, while performing interactive geometry manipulations would be one possibility to address this. Some KBE-augmented CAD systems already perform reasonably in this sense, although the quality and nature of the generated code is not yet at true KBE level. On the other side, the graphical interface of state-of-the-art KBE systems allows a very limited interaction with the generated geometry models, such that a fluid switch between code editing mode and interactive modeling is not yet at reach.

Independent of the aforementioned additional code generation capability, the existing language-based KBE systems can improve the feedback to user around geometry manipulation. For instance, they could give context-sensitive suggestions and guidance when some geometry operation fails.

11.4. Supporting web collaborative solutions and open source initiatives

When the first KBE system came into the market, the World Wide Web did not yet exist, nor was the concept of open source software widely known or appreciated. It is natural to expect new KBE systems to be geared toward their use and deployment via the Web [46]. The possibility to free the way to open source solutions, from one side, would relieve the end-user to return so often (with cash) to the “KBE vendor shop” and buy dedicated plug-ins (e.g., for graphics generation and visualization, or for the
manipulation of XML files). From the other side, it would relieve the KBE vendor from the burden of in house development of typically short life and buggy applications and extensions which may or may not align well with their core focus. Indeed, open source application and extensions have the potential for having fewer bugs than vendor supplied attempts, since a large number of peers have the possibility to bring improvements.

11.5. Enabling dynamic code → -- documentation generation

In order to allow developers accessing and understanding applications written by others, code documentation is extremely important; unfortunately producing and keeping updated quality documentation is as much useful as time-consuming. The black-box effect of unclerely structured and undocumented KBE applications, however, is a major point of concern. Any KBE application represents an investment. As such, the risk of making it unusable or too difficult to update just because the original developer has left the company without leaving proper documentation, can be as high as a showstopper for the diffusion of KBE in industry [18,54].

A KBE system that is able to autonomously produce descriptive documentation of its own application (possibly implementing standard visual modeling languages such as the UML [58],) would be more than welcome to all developers who, at least once, have struggled to debug or rework code written by others, or even by themselves. To this purpose, there are already some KBE systems, like GDL, that offer a so called Documentation Tool, to allow the automatic generation of HTML-based documentation from a set of KBE code files. Besides, underlying Common Lisp implementations already provide browsers and inspectors, which allow diagrammatical visualization of class definitions [59]. Although such capabilities are not yet included in the standard release of KBE systems such as GDL, the opportunity is there. It should be acknowledged that the instantiated product tree itself (Figs. 4 and 5) acts as diagram of has-a relationships, but without is-a relationships being immediately apparent. However useful, it is not the kind of visual representation offered, for example, by an industry standard diagrammatic language such as the UML. An example of UML product tree is provided in Appendix A, Fig. A2.

The automatic link code-documentation should actually work dynamically and in both directions. Engineers should have the possibility to agilely and interactively generate diagrams representing the structure and the design process of their products, having an interpreter active in the background that is able to generate at least the main structure of the KBE application code. Furthermore, it should be possible to visualize the structure of a given application and modify its code just by modifying its diagrammatic representation. The benefit would be twofold: first, the perceived obtrusive-ness of the programming language could be furthermitigated; second, KBE applications would lose their perceived black-box appearance, being the embedded knowledge properly exposed and formalized in a (standard) diagrammatic way. Preliminary demonstrators of automatic ICAD code generation, based on diagrammatic, UML-based, knowledge representation [15], were already developed within the MOKA project. Research in that direction is still ongoing [21], but no commercial KBE tool seems to offer this capability yet 7. In this direction, it is of interest the development of Grasshopper, the graphical algorithm editor developed to integrate with Rhino’s 3-D modeling tools, allowing generative CAD modeling without no knowledge of programming or scripting being required by designers [60]. Without any doubts, future KBE systems will benefit from any progress in the fields of knowledge modeling and storage frameworks development and the ability to integrate with them. Even further, the impact and diffusion of future KBE systems will benefit from the advances in the development of design rationale systems [61]. A combination of these systems with KBE would offer designers a powerful and more complete toolbox to enhance the level of automation in design, to improve the level of support for decision making, as well as a powerful, dynamic and transparent system for knowledge capturing and re-use.

Appendix A. An example of defpart definition

To further clarify the concepts presented in Section 6 and let the reader appreciate the easiness of reading a piece of KBE code, a sample of IDL code (the ICAD Design Language) is illustrated here. Fig. A1 shows a defpart operator used to define the hypothetical class ConventionalAircraft. The relative UML class diagram and object tree are provided in Fig. A2.

The mixin list includes the classes Aircraft and CostEstimationModule, from which ConventionalAircraft inherits. Hence, all the attributes and components of these two classes become also attributes and components of ConventionalAircraft.

Note that the two attributes horizontalTailSpan and verticalTailSpan used to define the Tail part, are neither defined as inputs nor attributes of ConventionalAircraft. It can be assumed they are inherited from the superclass Aircraft.

The CostEstimationModule, rather than a real superclass of which ConventionalAircraft would be a specialization, represents a class (not shown in the example) containing some kind of costs calculation procedure. Being included in the ConventionalAircraft mixin list, any instantiation of ConventionalAircraft will inherit the capability of computing costs.

ConventionalAircraft has been defined as aggregation of 4 classes, namely Fuselage, Tail, Wing and AircraftCog. It means that any instantiation of ConventionalAircraft will be composed of four (child) objects, of which the first three represent the main aircraft subsystems. The instantiation of AircraftCog, instead, would yield a non-geometrical object with the ability to compute the position of the aircraft center of gravity. This means that both geometrical and non-geometrical components can be heterogeneously structured in the object tree.

As specified using the command “Type” (see also Fig. A2-top), the classes Fuselage, Wing and AircraftCog are specializations of the classes Cylinder, WingGenerator and CogEstimationModule, respectively. The superclass of Tail, instead, is dynamically evaluated by means of a production rule. Accordingly, a Tail instantiation could result either in a “null object”, in case the attribute typeOfTail is evaluated to “tailless”, or in some other kind of tail, such as the conventional configuration assumed in the example of Fig. A2 (bottom).

Although not shown in the example, this KBE application is supposed to contain the definitions of Cylinder, WingGenerator, CogEstimationModule and some other class for the different types of tail. While the class Cylinder is actually one of the basic geometry classes predefined in ICAD (a so called ICAD geometry primitives), the others will have to be defined by the user by means of the defpart macro. As shown in the UML representation of Fig. A2-bottom, the object myTail contains two instantiations of the classes HorizontalTail and VerticalTail (although not shown in the example, both HorizontalTail and VerticalTail could be instantiations of two

7 It should be acknowledged that in May 2002, during the International ICAD User Group in Boston, KTI announced and presented ICAD Release 9, which was including a so called IDE (Integrated Development Environment) to help engineers developing ICAD applications by means of dragging and dropping predefined and user-defined classes on a canvas, to be connected and adjusted as needed. Release 9 included also a graphical user interface to allow interactive geometry picking, but after KTI acquisition by Dassault Systemes ICAD release 9 never came to the market.
specializations of the class WingGenerator. The same would apply for the object myWing). It follows that the object myTail is 1) parent of the two child objects myHorizontalTail and myVerticalTail, 2) one child of myAircraft, and 3) sibling of myWing and myFuselage.

The object tree shown at the bottom of Fig. A2 is the way a KBE system presents the modeled product to the user (actually a simplified version of the UML graph in the picture). Indeed, such has-part hierarchies are very familiar to engineers dealing with complex product configurations consisting of assemblies, subassemblies, components, subcomponents, parts and so on. The amount of hierarchical levels in the object tree generated by a state-of-the-art KBE system is actually unlimited.
Fig. A2. (Top) UML Class diagram for the ConventionalAircraft class showing inheritance and composition links. (Bottom) Object tree resulting from the instantiation of the ConventionalAircraft class.

References


