EDELANG: Environmental Design Language

Siniša Kolarić
School of Interactive Arts and Technology
Simon Fraser University
250-13450 102nd Avenue, Surrey, BC, V3T 0A3 Canada

March 14, 2009

Abstract

In this work we present EDELANG, a high-level modeling language for environmental design. EDELANG is a hybrid language inspired by existing high-level declarative languages and by ontologies. The user specifies the *what* of the design, and EDELANG then decides on the *how* in creating the final design, thus freeing the user from having to specify procedural details.

Introduction

In traditional societies everybody knows how to build; everyone builds for himself, and helps his neighbour build (Alexander 1979). The architectural language ("the pattern language") is known to everybody, and everyone can design a great building. Nowadays however, architects and designers in general have their own, specialized, privatized languages; and ordinary people have been robbed of their design intuitions by specialists. The common, communal and shared expertise on how to make great-looking and well-functioning buildings has been lost. To remedy this situation, Alexander calls for the rediscovery of these common shared languages, and to make the "patterns" of how to design great building explicit again, and aproachable by everybody.

In a similar vein, the main motivation for this work was to create a powerful modeling and programming language for environmental design which would enable even non-specialists to create beautiful, sound and usable architectural, product or engineering designs (or any other type of environmental designs), without having to deal with overly complex or expensive CAD packages. The general goal is to enable the users of EDELANG to specify *what* they want in a design to the desired level of detail, and EDELANG will do the rest for them, i.e. create the final design. The user would thus be freed from having to:

- Specify low-level details (like which bolts to use, or exact wall locations in milimeters) or
- Execute complex operations (like working through the GUI interface of CAD applications, or creating traditional blueprints) or
- Specify complex procedures, i.e. having to write complex scripts or code.

Thus EDELANG can, to a degree, be called a "virtual designer" technology; as with a real designer, the users specify *what* they desire in a design, and EDELANG then goes on to create the final artifact design.

Guiding Principles in Creating EDELANG

Our guiding principles in creating EDELANG were:

- EDELANG shall be a **formal and well-documented language**. As such, it comes with a:
  1. Formal specification,
  2. Reference compiler implementation which can generate 3D models (both volumetric models with materials, or conventional 2D triangle meshes with materials),
  3. 3D viewer,
  4. Rich web-based library of components, where volunteers around the world create, modify and share their own design components,
  5. Authoring tool (for new components and new design styles),
  6. Complete documentation,
  7. Tutorial, and
  8. Set of EDELANG code samples.
- EDELANG shall generate the **greatest possible effect for the smallest source code**. That is, the goal is for EDELANG programs to be concise and compact; what isn't specified, EDELANG generates, and decides upon, by itself. For example, a minimal EDELANG program that consists of just one keyword:

  ```
  house
  ```

  Even though it's minimal, this EDELANG program will create a full single-family house model ("default house") as shown in Fig. 1, generated by the implied set of default parameter values.

  Another minimal program which consists of one keyword:

  ```
  chair
  ```

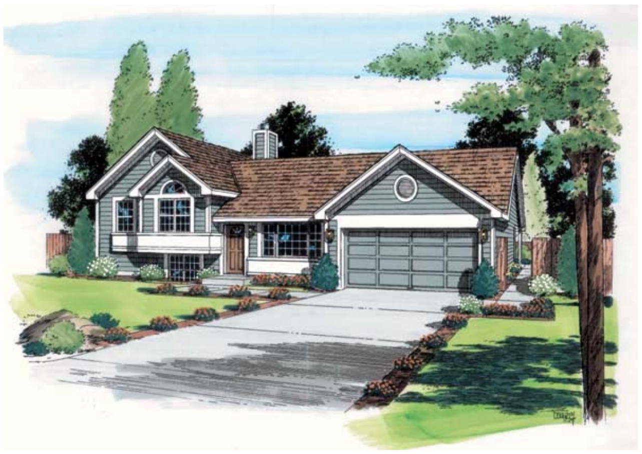

Figure 1: The "default" detached residence model produced by the minimal EDELANG program `house`. House design © ehouseplans.com.

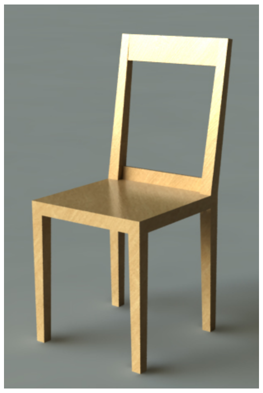

Figure 2: The "default" chair produced by the minimal EDELANG program `chair`. Chair design © ignis-fatuus.com.

This EDELANG program will create a simple chair model ("default chair") as shown in Fig. 2, generated by the implied set of default parameter values.

However, if the user wishes to customize the design, he can go deeper and develop an almost arbitrarily long EDELANG programs, overriding some or most of default values:

```
house( name="My house" )
{
  entrance
  kitchen
  living_room( area=300sqf )
  bedroom
  bedroom()
  space
  {
    sofa
    armchair
    armchair
    table
  }
}
```

In the EDELANG program above, a parameter `name="My house"` was specified and the body of keyword `house()` has been expanded and populated by a number of semantically rich keywords (`entrance`, `kitchen`, `living_room`, ...) specified by the user. The default value for area of one of these spaces (`living_room()`) has been overridden by the user-specified value of 300 square feet.

Also note that the parameter scope denoted by a pair of parentheses "(...)" is optional in EDELANG, as demonstrated by `bedroom` and `bedroom()`.

- EDELANG shall be a **high-level** hybrid (modeling + programming) language specifically developed for the purposes of environmental design. Our intention is to allow the EDELANG user to create environmental designs using high-level design concepts and constructs. By having concise and high-level language constructs, we avoid exponential decrease in productivity as the size of a program increases. According to (Brooks 1978), the total programming effort is

$$\text{programming effort} = K \times \text{instructions}^{1.5}$$

Thus the total effort expended in writing programs grows exponentially with the total number of instructions. By having more concise and higher-level instructions, we invest less effort in developing and maintaining EDELANG programs.

For example, if the user wants to create a kitchen, it is sufficient to type

```
kitchen
```

If the user wants to create a refrigerator in this kitchen, it is sufficient to type

```
kitchen { refrigerator }
```

- EDELANG shall be a **human-readable, user-friendly and expressive** programming language specifically developed for the purposes of environmental design. Differently from ontological languages which are machine-friendly and overly verbose (e.g. RDF or OWL), or are difficult to read or work with for average users (for instance, various logics), EDELANG code is meant to be readable, friendly and accessible even to non-programmers.
- EDELANG shall be a **declarative** programming language for environmental design, meaning it describes what an artifact is like, rather than how it is built up. Therefore, instead of specifying the control flow and a sequence of steps to execute, the user of EDELANG specifies the properties of the final design, and the computational intelligence built into the EDELANG compiler decides on what steps to execute. Of course, the behind-the-scene workings of EDELANG can be influenced by and are under the control of the user, either through parameters

passed to the compiler, or directly through EDELANG programs.

- EDELANG shall be a **constraint-based** programming language for environmental design, meaning it provides the user with the means to declare constraints between components of the design. Related to the fact of being a declarative language, EDELANG enables to user to specify and control the relations, called *constraints*, between components of the design solution. Examples of such constraints include "this room should be adjacent to the bedroom" or "an alcove should be in the northwestern corner of the kitchen". One example of EDELANG code containing a "distance" and an "opposite from" constraint is:

```
house
{
  room( id=A )

  room( id=B )

  room( id=C,
        opposite_from=A.southeast)

  distance( from=A, to=B, is=2m )
}
```

  In the program above, we stipulate that the distance between rooms A and B should be 2 meters, and that C should be opposite from the southeast side of A. During the compilation, EDELANG takes all the specified constraints into consideration and creates the final building design (i.e. configuration of spaces and rooms) accordingly.

- EDELANG shall allow **rapid prototyping** of environmental designs, thus leading to quick turn-around and fast evaluation of the same. For example, changing the relative positions of two rooms requires just a quick edit of the source code and recompilation of the corresponding EDELANG source file. Thus the following EDELANG program, which stipulates that space B should be to the left of A:

```
house
{
  space( id=A )

  space( id=B, opposite_from=A.left )
}
```

  becomes

```
house
{
  space( id=A )

  space( id=B, opposite_from=A.right )
}
```

  The source code above now stipulates that space B should be to the right of space A (more precisely, space B should be opposite from the right side of A). Recompiling the code, the resulting house model will have space B to the right of space A.

- EDELANG shall emphasise **separation of content, topology and geometry**. Conventional CAD packages mix content, topology and geometry in the same view or operation. In EDELANG, the user separately lists/specifies components, the topology (i.e. structure or connectedness) of these components, and references to geometry/shape. EDELANG compiler then generates the 3D model. However, if the user wishes so, he can override default rules for topology and geometry generation by specifying constraints, thus customizing the final design. For example, if we want to have a kitchen with a round floorplan instead of the default rectangular floorplan, we do not manipulate geometry directly, but specify a value for the parameter `floorplan` instead:

```
house
{
  kitchen( floorplan=circle )
}
```

System Architecture

This section describes the organization of the EDELANG software agent (i.e. of the compiler) and the dependencies between its parts.

Entities Produced During a Compilation Run

During a compilation run, EDELANG produces several intermediary entities.

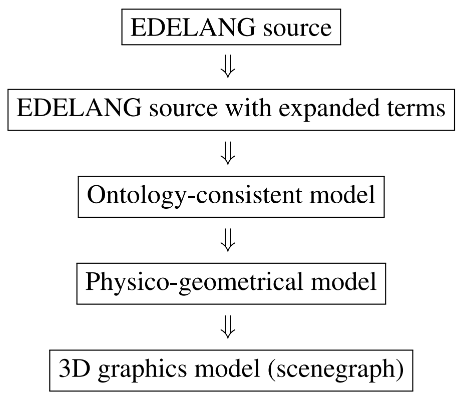
EDELANG source
EDELANG source with expanded terms
Ontology-consistent model
Physico-geometrical model
3D graphics model (scenegraph)

Figure 3: Entities produced during a compilation run.

Therefore, EDELANG compiler produces several entities in the process of compilation:

1. **EDELANG source**: this file contains the source of the EDELANG program, in text format.
2. **EDELANG source with expanded terms**: this intermediary file contains the full EDELANG model, after all high-level container concepts have been expanded.
3. **Ontology-consistent model**: this intermediary file contains a full formal model of the building, consistent with the corresponding ontology for buildings.

4. **Physico-geometrical model**: this intermediary file contains a full physics-based model of the building being programmed.
5. **3D graphics model (scenegraph)**: the typical output at the end of the pipeline. However, other outputs are possible, like for example a raster image of the building.

Phases in a Compilation Run

During a compilation run, EDELANG compiler goes through the following phases:

1. **EDELANG source ⇒ EDELANG source with expanded terms**: this phase expands high-level concepts in the input EDELANG source. For example, `kitchen()...` gets expanded into the full subtree of all the components contained in this particular kitchen.
2. **EDELANG source with expanded terms ⇒ ontology-consistent model**: this phase translates the full EDELANG model into a full, formal building model which is consistent with its corresponding ontology.
3. **Ontology-consistent model ⇒ physico-geometrical model**: this phase translates the full formal model of the building, consistent with the corresponding ontology, into the full physics-based model (which also includes the geometry of the building).
4. **Physics-based model ⇒ 3D graphics model (scenegraph)**: this phase translates the full physics-based model of the building into the corresponding scenegraph, ready to be rendered on display. This phase could, alternatively, render the physics-based model into another type of file, for example into a raster image.

A Session of EDELANG

In this section, we work through various EDELANG sample programs in order to give the readers a better feeling of the language.

The following EDELANG program will create a full single-family house model ("default house") as shown in Fig. 1, generated by the implied set of default parameter values:

```
house
```

Since no parameters have been specified for this particular keyword (`house`), the set of *default parameters* was assumed. For example, the house style was set to the default house style (`us_ranch_2000`), and the total area of the house was set to default house area for this style, namely 2,330 square feet[1]. The program above is therefore equivalent to

```
house( style=us_ranch_2000, area=2330sqf )
```

If we want to retain the style (i.e. default style) for this house, but want to increase the total area of the house to 3,000 square feet, we write:

[1]According to the National Association of Home Builders (www.nahb.com), the average home size in the United States was 2,330 square feet in 2004.

```
house( area=3000sqf )
```

or equivalently

```
house( style=us_ranch_2000, area=3000sqf )
```

If we now want to change the style of the house to bungalow, but still want to retain the total area of 3000 square feet, we write:

```
house( style=us_bungalow_2000, area=3000sqf )
```

The recommended convention for naming styles is <*region-style-year*>, where *region* represents the style's characteristic geographic region, *style* is the name of the style, and *year* represents this particular style's most prolific year. However, this convention is not mandatory so other style names are possible, like for example `paladio_emo` which is based on the style used by classic architect Andrea Palladio when designing one specific building, namely Villa Emo.

It should be noted that power users of EDELANG can create their own styles, and save them into files which can afterwards be utilized by EDELANG. The author of a style can then either share his style with other EDELANG users, or not share at all (for example, if the style creator wants to preserve his/her competitive advantage in creating unique house designs). For example, if a user authored a style called *my_own_style*:

```
house( style=my_own_style, area=3000sqf )
```

The program above will create a house with the total available area of 3,000 square feet, using the style `my_own_style`. Note that a separate tool will be provided in the reference implementation and which will allow EDELANG users to author their own styles.

We will now say more about the inner structure (number of levels/stories, floorplan and room configuration, and aesthetics/appearance) of generated house designs. As we said above, the following minimal program:

```
house
```

generates a model with the default style. However, the default style also dictates the *default inner structure* of the house. In this case, the current style is set to the default style (`us_ranch_2000`), which is characterized by:

- A single (ground) story
- Open, rambling room layout, with few inner partitions
- Built using natural materials: wood, brick
- Minimal decoration
- Asymmetry
- Low pitched gable roof; long roofline with large overhanging eaves
- Large windows
- Patio
- Attached side garage.

If we now change the style to *split level*, as practiced in US in 2000, we write:

```
house( style=US_splitlevel_2000 )
```

and obtain a model with the following structure and appearance:

- Three stories:
  - Basement level: large family room, utility rooms, garage.
  - Main level: family room, living room, dining room, and kitchen.
  - Top level: bedrooms, bathroom.
- Minimal decoration
- Pitched gable roof
- Large windows.

In the following, we will say more about customizing default designs. For example, the following minimal EDELANG program

```
house
```

will generate a house model with default style, default inner structure and default appearance. However, the user of EDELANG will frequently want to customize this design further. For example, we could override the default number of stories (which is set to 1 when we use the default `us_ranch_2000` style) by writing

```
house( nr_stories=2 )
```

Therefore, in this case EDELANG will try to respect all the characteristics of the current style, all the while configuring all the spaces into two stories, not just one story as dictated by the current style `us_ranch_2000`.

Also, if an EDELANG user wants to have a different set of rooms/spaces in the design, he could begin by writing:

```
house
{}
```

As we can see, the program above opened a scope using a pair of parentheses {}, and this produces the effect of *removing all the default spaces/rooms* from the final generated model, while still retaining the outward appearance and total available area of the house. If we now write

```
house
{
  kitchen
}
```

the generated house model will still retain the outward appearance and total available area of the house, but with the addition of one kitchen. This kitchen will have the default area as defined by the style, and the rest of the story will be empty. Since the keyword *kitchen* carries rich semantics, EDELANG will also automatically create all the content relevant to a kitchen (like plumbing, etc.) in the final house model. If the user wants to create a simple space, however, the keyword `space` should be used.

Furthermore, subparts of the house design can have their own styles. For example, in the last EDELANG program above, the component `kitchen` assumed the style inherited from its parent component `house`, and this style was equal to `us_ranch_2000`. However if we specify

```
house
{
  kitchen( style=farmkitchen )
}
```

the kitchen will assume its own style, that is of "farm kitchen", while the rest of the house model retains the ranch style. All the sub-components within kitchen (like stoves, sinks, cabinets, etc.) will then also inherit this "farm kitchen" style and will be chosen accordingly so that their appearance and function is compatible with the farm style. Again, the rest of the house model will retain the `us_ranch_2000` style.

Prior Work

Configuration Design

Configuration design is a kind of design where a fixed set of predefined components that can be interfaced (connected) in predefined ways is given, and an assembly (i.e. designed artifact) of components selected from this fixed set is sought that satisfies a set of requirements and obeys a set of constraints (Mittal and Frayman 1989).

The design configuration problem consists of the following three constituent tasks (Levin 2009):

1. Selection of components,
2. Allocation of components, and
3. Interfacing of components: design of ways the components interface (connect) with each other.

Various techniques to solving the design configuration problem exist, and include (Levin 2009): the shortest path problem; versions of basic multiple choice problem and multicriteria multiple choice problem; multicriteria decision making approaches; traditional morphological analysis methodology and its modifications; multipartite graph clustering; hierarchical morphological design approach based on morphological clique problem; parametric design; heuristics for component set identification problem; evolutionary approaches (genetic algorithms, etc.); multiagent approaches; approaches based on fuzzy sets; constraint-based methods including composite constraint satisfaction problems; ontology-based approaches; AI techniques; and design grammars approaches (e.g., multidisciplinary grammar approach that includes production rules and optimization, graph grammar approach).

Types of knowledge involved in configuration design include (Wielinga and Schreiber 1997):

- Problem-specific knowledge:
  - Input knowledge:
    - ∗ Requirements

    * Constraints
    * Technology
  - Case knowledge

- Persistent knowledge (knowledge that remains valid over multiple problem solving sessions):
  - Case knowledge
  - Domain-specific, method-independent knowledge
  - Method-specific domain knowledge
  - Search-control knowledge

Semantic Web

Traditional HTML pages are designed to be read by people, not parsed by machines. Based on the existing XML standards, the W3C consortium has created several specifications geared towards allowing computers to "understand" the content accessible over the Web. Of these, the most important ones are the Resource Description Framework (RDF), and the Web Ontology Language (OWL).

RDF is a general-purpose language for representing information in the Web. It describes *resources* (these include any imaginable entity – resources accessible on the Web, tangible objects, abstract entities, ...) in terms of named properties and values. The RDF vocabulary description language, RDF Schema (RDFS), describes vocabularies used in RDF descriptions; vocabularies define classes of resources, properties, and relationships between classes. For example, to compare RDF and RDFS with traditional object-oriented (OO) systems, in OO systems the developer of a class controls what properties a class will have; in RDF and RDFS, anyone is able to define a new property of a class. Therefore, one could say that OO systems are resource-centric and centralized, while RDF and RDFS are property-centric and decentralized.

OWL was designed in order to overcome the limited expressiveness of RDFS. The purpose of OWL is to provide a family of languages that can be used to describe ontologies (that is, classes, and relations between them). OWL is more powerful than file structuring capabilities such as DTDs and XSD. There are three versions of OWL that contain constrained subsets of the language and that may be used for various purposes:

- OWL Lite is a simple language intended to satisfy users needing a classification hierarchy and simple constraint features.
- OWL DL uses concepts from formal Description Logic (hence the DL designation). It includes the complete OWL vocabulary, but some constraints are placed on usage.
- OWL Full contains all the OWL language constructs and provides free, unconstrained use of RDF constructs.

OWL has a well-defined syntax and semantics, and support for automated reasoning, through the use of so-called *reasoners*. For example, in OWL one can conduct automated tests for class membership of an instance, test equivalence of classes, check for errors both in the ontology and associated knowledge bases, or check for unintended relationships between classes. These functionalities are crucial when integrating ontologies from different domains, or when dealing with very large OWL code bases.

Domain-Specific Languages

Domain-specific languages (DSLs), also called *little languages* (Bentley 1986), serve to develop solutions by using concepts characteristic to a specific domain. In their respective domains of application, DSLs offer significant gains in expressiveness and ease of use, compared with general purpose programming languages such as C, C++, Java, Ruby and Python. Examples of well-known and successful DSLs include SQL, Matlab, HTML, TeX/LaTeX and VHSIC Hardware Description Language (VHDL).

DSLs that are successful in their respective niches are characterized by the following principal factors (Sprinkle et al. 2009):

- They satisfy the domain requirements,
- Restrict the user to input only the parameters characteristic of the domain, and
- Give the users easy access to constructs used in the domain.

In the ideal case, a DSL utilizes domain constructs with semantics that are as close to the original semantics as is possible, thus allowing the user to work directly with the concepts from the domain. The resulting DSL source code is thus on a higher level as compared to source codes written general-purpose programming languages, and represent simultaneously (Sprinkle et al. 2009) the:

1. Design,
2. Implementation, and
3. Documentation

of the artefact being modelled or programmed. EDELANG can thus be viewed as a family of DSLs for developing designs in a particular sub-domain of environmental design.

Conclusion and Future Work

We presented EDELANG, a very high-level modeling and programming language for environmental design. As distinguished from current CAD tools and packages, EDELANG puts much more emphasis on allowing the user to specify *what* to include in the final building design, instead of *how* to create the design.

The guiding design principles for EDELANG are:

- Formal specification.
- Greatest possible effect (i.e. a complete building design) for as little source code as possible. What isn't specified in the source code, is decided upon by EDELANG.
- Human-readable and user-friendly language, thus usable even by non-programmers.
- Declarative, goal-oriented and constraint-based programming language.

- Support for rapid prototyping of environmental designs.
- Separation of content, topology, design and constraints.

Within the classic "design problem space" vs. "design solution space" dichotomy, the version of EDELANG proposed in this work represents a point in the solution space. Future work includes extending EDELANG to cover the design problem space as well, by including entities characteristic of this space such as 1) mission statements, 2) design goals, 3) early-phase design constraints and 4) functional and non-functional design requirements.

References

Alexander, C. 1979. *The Timeless Way of Building*. New York, USA: Oxford University Press.

Bentley, J. 1986. Programming pearls: little languages. *Commun. ACM* 29(8):711–721.

Brooks, Jr., F. P. 1978. *The Mythical Man-Month: Essays on Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Levin, M. S. 2009. Combinatorial optimization in system configuration design. *Autom. Remote Control* 70(3):519–561.

Mittal, S., and Frayman, F. 1989. Towards a generic model of configuration tasks. In *Proceedings of the 11th IJCAI*, 1395–1401. San Mateo, CA, USA: Morgan Kaufman.

Sprinkle, J.; Mernik, M.; Tolvanen, J.-P.; and Spinellis, D. 2009. What kinds of nails need a domain-specific hammer? *IEEE Software* 26(4):15–18. Guest Editors' Introduction: Domain Specific Modelling.

Wielinga, B., and Schreiber, G. 1997. Configuration-design problem solving. *IEEE Intelligent Systems* 12:49–56.

Appendix: EDELANG specification

EDELANG is case insensitive.

EDELANG Grammar Rules (Extended Backus-Naur Form)

program $\longrightarrow$ *declaration-list*
declaration-list $\longrightarrow$ *declaration-list* | *declaration*
declaration $\longrightarrow$ *fun-declaration*
fun-declaration $\longrightarrow$ *ID (params) compound-stmt*
params $\longrightarrow$ *param-list* | *empty*
param-list $\longrightarrow$ *param-list, param* | *param*
param $\longrightarrow$ *parameter = expression*
parameter $\longrightarrow$ *ID*
expression $\longrightarrow$
compound-stmt $\longrightarrow$ { *statement-list* }