Design Constraint Systems - A Generative Approach to Architecture

Dietrich Bollmann and Alvaro Bonfiglio

Design Constraint Systems - A Generative Approach to Architecture

Dietrich Bollmann and Alvaro Bonfiglio

Generative Architectural Design permits the automatic (or semiautomatic) generation of architectural objects for a wide range of applications, from archaeological research and reconstruction to digital sketching. In this paper the authors introduce design constraint systems (DCS), their approach to the generation of architectural design with the help of a simple example: The development of the necessary formalisms to generate a family of architectural designs, i.e. simple houses and pagodas. After explaining the formal system the authors introduce an approach for the generation of complex form based on the application of transformations and distortions.

Architecture is bound by the constraints of physical reality: Gravitation and the properties of the used materials define the limits in which architectural design is possible. With the recent development of new materials and construction methods however, the ways in which form and physics go together get more complicated. As a result, the shapes of architecture gain more liberty, and more and more complex shapes and structures become possible. While these advances allow for new ways of architectural expression, they also make the design process much more challenging. For this reason new tools are necessary for making this complexity manageable for the architect and enable her to play and experiment with the new possibilities of complex shapes and structures. Design constraint systems can be used as tool for experimentation with complex form. Therefore, the authors dedicate the final part of this paper to a concise delineation of an approach for the generation of complex and irregular shapes and structures.

While the examples used are simple, they give an idea of the generality of design constraint systems: By using a two-component approach to the generation of designs (the first component describes the abstract structure of the modelled objects while the second component interprets the structure and generates the actual geometric forms) and allowing the user to adjust both components freely, it can be adapted to all kind of different architectural styles, from historical to contemporary architecture.

1. INTRODUCTION

While first only seen as a more powerful substitute for tools traditionally used by architects like the drawing board or desk calculator, the computer is applied more and more to the generation of design options. Parametric design [1], shape grammars [2] or the creation of three-dimensional structures by structural optimization strategies [3, 4] are examples of this trend in architecture. In this paper we recombine ideas from these research fields – the variation of designs with parameters, the use of grammars for the description of design families and the idea of searching design spaces with optimization strategies –, integrate them with type constraint systems (TCS) [5], a formalism widely used in Computational Linguistics [6,7], and propose design constraint systems, a formalism for the generative description of architecture.

Design constraint systems (DCS) allow for the generative description of existing types and styles of architecture, like Greek temples, Japanese pagodas or African villages [8], as well as the experimentation with new architectural rule systems and styles.

DCS are organized in two components: A system of constraints, formulated as *type constraint system* (TCS), describing the abstract rules underlying the architectural style that is to be modelled and a formal language for the description and generation of two or three-dimensional shapes. The first component is called *design grammar* and the second one *design description language*. Design grammar (DG) and design description language (DDL) together define the *design space*, the set of designs corresponding to the type or style of architecture to be modelled.

Design spaces (DS) can be very large and therefore, given a certain design problem, it can be difficult to find the most adequate designs. Strategies to search design spaces for good designs therefore are another important part of design constraint systems.

1.1. Basic Conditions

Design constraint systems are based on four basic hypothetical conditions. Even if formulated in a general way, they are not meant as general statements about architecture but as conditions for the applicability of design constraint systems.

The first condition or assumption is that architectural artefacts can be described by an abstract structure called design structure (DS). Depending on the class of modelled objects, design structures can be very different. In the simplest case, design structures describe the compositional structure of a building, i.e. the parts it is assembled from, their relation to each other and their attributes like size or location. In more complex cases, the design structure might describe how volumetric primitives have to be generated, modified and combined in order to result in the shape characteristic for the building described. A verbal description like "The house consists of a

pyramid shaped roof with uplifted corners posed on top of a base floor in form of a cuboid." is very similar to a design structure: The parts of the house are named (house, roof, pyramid, floor, cuboid), necessary operations on them are specified (uplifted corners) and the relation to each other is given as well (consists of, posed on top of).

In our system design structures are formalized as *typed feature structures*, which are part of a powerful formalism widely used in computational linguistics. Typed feature structures will be explained in greater detail below.

The second general assumption is that the relation between design structure and architectural form can be described by a design description language (DDL). A design structure by itself is only a syntactic construct and doesn't say much about the actual shape. Only by interpreting the design structure and translating it into a two or three-dimensional form, the structure gets an actual meaning. This interpretation or translation of design structures into two- or three-dimensional shapes is formalized by the *design description language* (DDL). The ensemble of forms, which can be described and generated by the design description language, is called *design universe*.

When dealing with a family of graphic structures assembled from straight lines for example, lists of line descriptions made from attributes like line length, width, colour and position could be used as design description language. The corresponding design universe would be the set of all combinatorially possible drawings made from straight lines. A more complex design description language could be made from token describing a set of volumetric primitives and operations on them. Design languages like this will be introduced later for the description of simple houses, pagodas and distorted structures. Depending on the architectural domain and style, different design description languages are adequate.

The third assumption considered in this paper is that the set of design structures corresponding to a certain architectural type or style can be formalized by a design grammar (DG). A design description language allows for the description of arbitrary objects in the design universe. But in relation to a specific architectural style only a specific subset of these objects is of interest. In order to differentiate the number of structures describing instances of this style from the set of combinatorially possible structures, *design constraints* organised in a *design grammar* (DG) are used. The design grammar together with the design description language defines the design space (DS), the family of designs described by the design constraint system.

Finally, the fourth assumption or hypothetical condition is that design spaces can be searched for appropriate designs by evolutionary strategies. A design structure together with its generation history - the sequence of steps used to generate it from the design grammar - can be seen as a kind of "genetic code" of the corresponding design. By gradually modifying this code or combining it with those of different designs, the design space can

be searched for appropriate designs. The modification and combination of design structures and their generation history are similar to the processes of mutation and recombination in genetics and the equivalent procedures used in the field of evolutionary computation [9].

It is not always necessary to search the design space for good solutions: If the design constraints are very specific, the design space might only contain a small number of equally suitable designs. Sometimes design constraints are so specific that only one design fulfils them. The pagoda grammar presented later in this paper is an example for the latter, while the distorted pagoda introduced after is an example for a design constraint system, which relies on efficient search strategies.

In sum, combining those four assumptions we get to the following conclusion: Architectural styles or types fulfilling the four mentioned assumptions can be formalised by design constraint systems, which are composed from two components: A design grammar and a design description language. Evolutionary strategies allow searching the formalised family of design for suitable designs.

After this short introduction into the assumptions underlying design constraint systems, we will now have a look at the framework used to formalize our approach.

2. THE FORMAL FRAMEWORK

The development of systems with a certain complexity often is iterative: starting from a set of simple case studies, a first version of the system is designed. Applying it to more complex problems reveals where the system can be extended and leads to a next version. This process is repeated until a system results, which is able to successfully deal with the class of problems in question.

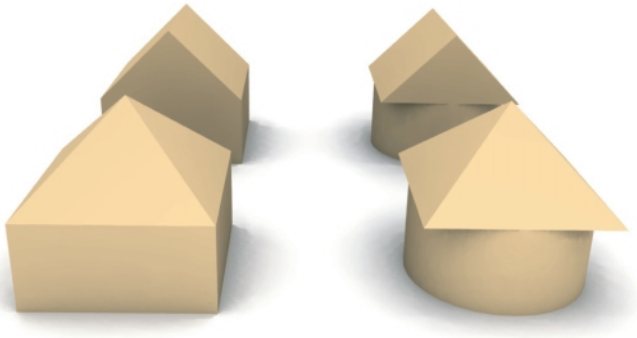
When explaining the resulting system, a similar iterative approach often is the most adequate. Our introduction into design constraint systems therefore proceeds in iterations as well: Starting with a simple case study and the necessary formalism to model it, we stepwise extend our domain and introduce the formal tools necessary to model it.

2.1. First Iteration: Simple Houses and Type Constraint Systems (TCS)

Our first example starts from a field most people are expert in: Toy blocks. Let us introduce our first example domain, four simple houses (see Fig. 1).

Looking at them closer reveals, that they represent all possible combinations of two building blocks used for the roofs with two others used as base (Fig. 2).

◀ Figure 1. Four simple houses.



◀ Figure 2. Combinations of roof and base volumes.

For our example therefore only four building blocks or, as we will call them from now on, *geometric primitives* are necessary: *cuboid, cylinder, prism* and *pyramid* (Fig. 3).

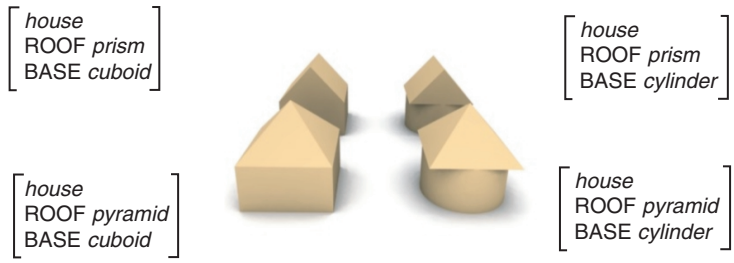
Using these primitives we can describe our houses with the following structures, specifying which primitives have to be used for the base and the roof in order to form the respective house (Fig. 4).

We call these structures *typed feature structures* (TFS) or, as they represent a design, *design structures* (DS).

The use of this kind of combinatorial structures for the representation and description of some entity has a long tradition in artificial intelligence, knowledge representation and computational linguistics. The *head-driven phrase structure grammar* (HPSG) for example [6, 7], a generative grammar

◀ Figure 3. Geometric primitives: *cuboid, cylinder, prism* and *pyramid*.

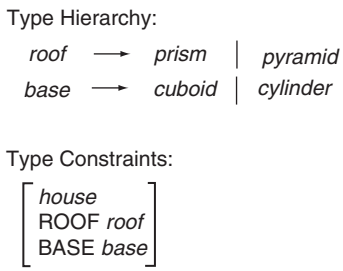
► Figure 4. Houses with design structures.



theory for natural language, is based on type feature structures as well and has a very active research community. *Typed feature logics* [5], the formalisms it is based on therefore is well studied. Making use of these research results, we based our system on *type constraint systems* (TCS) [5], a variant of the same formal framework, which we adapted and extended for the modelling of architecture.

The four structures of our example are easily enumerated. But in the case of more complex domains it would not only be cumbersome to list all possible structures explicitly, but such a list also would be highly redundant. Rather than listing these structures therefore one by one, we abstract their underlying regularities and describe them with a set of rules, which avoid the redundancy and still allow the generation of all structures (Fig. 5).

► Figure 5. Type constraint system for the generation of the design structures for simple houses.



The rules in the upper part are called *type subsumption rules* and define a *type hierarchy* while the structures in the lower part - there is only one in our simple example - are called *type constraints*. Both together form a *type constraint system* (TCS), which, as already mentioned, corresponds to one of the two components our system consists of.

Type constraint systems are complex and a detailed explanation of the theory underlying them is outside of the scope of the current paper. Let us however give an approximate idea by informally explaining the core concepts of type constraint systems. A more detailed explanation of TCS can be found in [5] and [10].

2.2. Types and Type Hierarchies

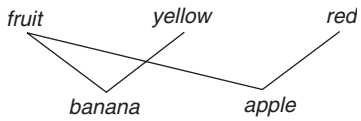
Types are the basic token on which type constraint systems are built. They can be understood as concepts used to name the entities of the modelled domain. Types are ordered and form a type hierarchy, which can be represented as a graph - here with the more general types higher than the more specific ones (Fig. 6).

In the case of our simple example domain, the types *prism*, *pyramid*, *cuboid* and *cylinder* denote atomic entities, i.e. the respective geometric primitives, while the type *house* is used to name the complex units, which can be built from them. In order to distinguish those elements, which are used as roofs from those used as base, the more general types *base* and *roof* are introduced. The most general type is called *top* and written with the symbol T. A type with subtypes can be understood as a concept which subsumes the more specialised concepts represented by its subtypes: Taking our type hierarchy as basis, a roof is either a prism or a pyramid and, the other way around, all prisms and pyramids are roofs.

In order to save place we often define type hierarchies by enumerating the direct subsumption relations as shown in the upper part of Figure 5, rather than by drawing them explicitly. In this notation, types are connected with their direct subtypes with an arrow and alternative subtypes are listed with a vertical line between them.

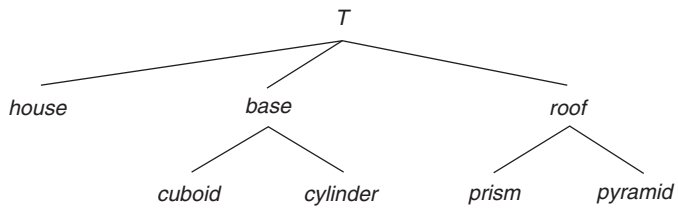
Type hierarchies define an order on types, called *subsumption order*. We say that more general types higher in the hierarchy *subsume* more specific ones lower in the hierarchy and directly or indirectly connected to them by down-pointing edges.

Types can be consistent or inconsistent: Types with common subtypes are consistent, while those without common subtypes are inconsistent. In the following hierarchy for example



(1)

the types *fruit* and *yellow* have the common subtype *banana* and therefore are consistent; the types *yellow* and *red* however are inconsistent as they don't have a common subtype.



◀ Figure 6. Type hierarchy for simple houses.

In order for a type hierarchy to be well defined, for all consistent types there has to be a unique most general common subtype. The most general common subtype of two consistent types is also called their *unifier* and the function, which selects it, when existing, is called *unification*. From a semantic point of view, the unifier of two types corresponds to the conjunction of their meaning. In the world described by the last example hierarchy for example, all yellow fruits are bananas and all bananas are yellow and fruits.

2.3. Typed Feature Structures (TFS)

In our example we used single types like *cuboid* and *pyramid* to represent the respective geometric primitives. For more complex artefacts as the houses in our example, however, we used structures like the following one containing more than one type:

$$\begin{bmatrix} house \\ ROOF\ roof \\ BASE\ base \end{bmatrix} \tag{2}$$

Single types and these structures together are called *typed feature structures* (TFS), with the single types being called *atomic* typed feature structures[1], and those combining more than a single type *complex* typed feature structures.

Let us look at the parts of TFS in more detail: The type written in the upper left corner is the *root type* of the structure. The token written with capitals are called *features* and the elements to their right feature *values*. A feature together with its value is called *feature-value pair*. In case of the given example structure with the root type *house*, there are two feature-value pairs: ROOF *roof* and BASE *base*. Feature values are atomic or complex typed feature structures themselves. In the following structure for example, the feature CONTENT in the outer structure has a complex value while the same feature in the inner structure representing its value has a simple one:

$$\begin{bmatrix} box & \\ CONTENT & \begin{bmatrix} box & \\ CONTENT & pearl \end{bmatrix} \end{bmatrix} \tag{3}$$

A chain of features like the one formed by the two CONTENT features is called a *path* and written with vertical lines connecting the involved features: CONTENT|CONTENT.

[1] In this context simple types actually can be seen as an abbreviation for typed feature structures with zero features. *root* for example corresponds to the structure [*root*].

Values can also be shared between different features. In this case the values are co-indexed with a small number written in a box, a so-called *tag*:

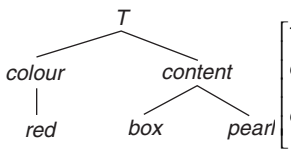
$$\begin{bmatrix} box & \\ COLOUR & \boxed{0}\ red \\ CONTENT & \begin{bmatrix} box & \\ COLOUR & \boxed{0} \\ CONTENT & pearl \end{bmatrix} \end{bmatrix} \quad (4)$$

In this structure the paths COLOUR and CONTENT|COLOUR share the same colour *red*, describing the colour of the two interlaced boxes.[2]

The subsumption relation defined on types can be extended to TFS: A TFS *A* subsumes another TFS *B* if and only if:

- All paths defined in *A* are defined in *B* as well.
- The root types of the values of all paths in *A* subsume the root types of the values of the respective paths in *B*.
- All paths sharing their values in *A* share their values in *B* as well.

Taking the type hierarchy on the left as basis, the structure in the middle subsumes the structure on the right, as can be easily verified by comparing the structures to the three criteria for the subsumption of TFS.



$$\begin{bmatrix} T & \\ COLOUR & \boxed{0}\ colour \\ CONTENT & \begin{bmatrix} content & \\ COLOUR & \boxed{0} \end{bmatrix} \end{bmatrix} \sqsupseteq \begin{bmatrix} box & \\ COLOUR & \boxed{0}\ red \\ CONTENT & \begin{bmatrix} box & \\ COLOUR & \boxed{0} \\ CONTENT & pearl \end{bmatrix} \end{bmatrix} \quad (5)$$

As types, TFS can be consistent or inconsistent. For every two consistent TFS a unique most general TFS, called *unifier* exists, which is subsumed by both original structures. The function, which selects the unifier, is called *unification* as before in the case of types. Finally, from a semantic point of view, again the unifier corresponds to the conjunction of the meaning of the two TFS unified.

The unifier of two TFS *A* and *B* can be constructed algorithmically by defining a new TFS C as follows:

- *C* contains all and only those paths defined in *A* or *B*.
- For paths defined in both original structures, *C* takes the unifier of the root types of their values as root type; for paths defined in only one of the original structures the type found in this structure is used.

---

[2] Feature values, which share their values, are the reason why typed feature structures represent graphs rather than trees. They also make it possible to formulate circular structures:

$$\boxed{0}\begin{bmatrix} lie & \\ ARG & \boxed{0} \end{bmatrix}$$

- *C* shares the values for all paths for which the values are shared in the original structures *A* or *B*, with the unifier of the root types of the values of the respective paths in the original structures as root type.

The following figure shows examples for the different cases occurring during the unification of TFS. The little u-like Symbol represents the unification operation.

$$1.\ \begin{bmatrix} basket \\ FRUIT\ fruit \end{bmatrix} \cup \begin{bmatrix} basket \\ FRUIT\ red \end{bmatrix} = \begin{bmatrix} basket \\ FRUIT\ apple \end{bmatrix}$$

$$2.\ \begin{bmatrix} basket \\ F1\ banana \end{bmatrix} \cup \begin{bmatrix} basket \\ F2\ apple \end{bmatrix} = \begin{bmatrix} basket \\ F1\ banana \\ F2\ apple \end{bmatrix} \qquad (6)$$

$$3.\ \begin{bmatrix} basket \\ F1\ fruit \\ F2\ red \end{bmatrix} \cup \begin{bmatrix} basket \\ F1\ \boxed{0}\ T \\ F2\ \boxed{0} \end{bmatrix} = \begin{bmatrix} basket \\ F1\ \boxed{0}\ apple \\ F2\ \boxed{0} \end{bmatrix}$$

Typed feature structures are used as general representation of information in type constraint systems. They are also used to represent type constraints (TC) as the one for type *house* shown in the lower part of the type constraint system displayed in Figure 5. A type constraint determines the set of obligatory features defined for its root type and the set of values allowed for them. The type constraint

$$\begin{bmatrix} house \\ ROOF\ roof \\ BASE\ base \end{bmatrix} \qquad (7)$$

for example can be read as follows: A house has two features: ROOF and BASE . The feature ROOF has a value, which is subsumed by the type *roof*, while the feature BASE has a value, which is subsumed by *base*.

Type constraint systems, while themselves finite, can be used to define potentially infinite sets of typed feature structures, called *solutions*. A solution is a typed feature structure, which fulfils all applicable type constraints and only contains types without further subtypes. A TFS fulfils a type constraint *C* for type *t* if all its substructures with a root type subsumed by *t* are subsumed by *C* as well. The structure

$$\begin{bmatrix} house \\ ROOF\ pyramid \\ BASE\ cuboid \end{bmatrix} \qquad (8)$$

for example is a solution: None of the types *house*, *pyramid* and *cuboid* it is assembled from has further subtypes and the type constraint for type *house*, the only one which can be applied, is fulfilled.

In DCS the solutions correspond to the design structures which are interpreted by the design language in order to generate the designs. In this paper we therefore use the terms solution and design structure interchangeably.

In order to get the solutions for a certain design problem, or for simply enumerating all possible solutions of a TCS, *queries* are used. A query is formulated as TFS as well. The structure

$$\begin{bmatrix} \textit{house} \\ \textit{ROOF pyramid} \end{bmatrix} \tag{9}$$

for example can be used to query the system for all solutions describing houses with pyramid shaped roofs.

A type hierarchy together with the corresponding set of type constraints defines the set of solutions for a query. But in order to actually obtain these solutions, we need a mechanism, which constructs and enumerates them for a given query and TCS: A *resolution algorithm*.

The working of the resolution algorithm can be deduced from the definition of a solution: We have to iteratively replace the types used in the query with one of their subtypes and apply the matching type constraints by unification until the resolution state corresponds to a solution. As normally there are different paths in the current resolution state which can be resolved next and types in most cases have more than one subtype, different selections of the resolution path, subtype and type constraint in general result in different solutions - if not, when a type constraint is not consistent with the resolution state, in no solution at all. Let us look at an example in order to get a better idea of the resolution procedure:

We now know enough to resolve queries related to the TCS shown in Figure 5, which here is repeated for simpler reference as Figure 7:

Let us enumerate the design structures of all simple houses, which can be generated with this TCS. We start from the most general query for simple houses, the type *house*:

$$\textit{house} \tag{10}$$

Type Hierarchy:

roof ⟶ *prism* | *pyramid*

base ⟶ *cuboid* | *cylinder*

Type Constraints:

$$\begin{bmatrix} \textit{house} \\ \text{ROOF } \textit{roof} \\ \text{BASE } \textit{base} \end{bmatrix}$$

◀ Figure 7. Typed constraint system for simple houses (repeated).

The type *house* has no subtypes. So the only possible next resolution step consists in the application of the type constraint for type *house*. The resulting structure has the same form as the type constraint itself:

$$\begin{bmatrix} house \\ ROOF\ roof \\ BASE\ base \end{bmatrix} \tag{11}$$

The values of both features, ROOF and BASE, have subtypes. We use *pyramid*, the second subtype of *roof*, and substitute it for the original one at path ROOF:

$$\begin{bmatrix} house \\ ROOF\ pyramid \\ BASE\ base \end{bmatrix} \tag{12}$$

Doing the same at path BASE using *cuboid*, the first subtype of *base*, we obtain the following resolution state:

$$\begin{bmatrix} house \\ ROOF\ pyramid \\ BASE\ cuboid \end{bmatrix} \tag{13}$$

None of the types in this structure has further subtypes, and all type constraints - in this simple example there is only one for type *house* - have been applied. The structure therefore is our first solution.

Trying all other possible combinations of subtypes, four solutions can be generated:

$$\begin{bmatrix} house \\ ROOF\ prism \\ BASE\ cuboid \end{bmatrix} \begin{bmatrix} house \\ ROOF\ prism \\ BASE\ cylinder \end{bmatrix} \begin{bmatrix} house \\ ROOF\ pyramid \\ BASE\ cuboid \end{bmatrix} \begin{bmatrix} house \\ ROOF\ pyramid \\ BASE\ cylinder \end{bmatrix} \tag{14}$$

They correspond to the example structures of simple houses in Figure 4, from which our introduction into type constraint systems started.

In this simple example, the resolution procedure was straightforward. But we can see already, that in the general case the number of possible choices for the individual resolution steps can be large resulting in an even larger number of possible resolution orders. The number of possible strategies for the selection of the next resolution step, the path, subtype or type constraint to apply, therefore has to be large as well.

Depending on the type hierarchy and the type constraints, the solution space of a TCS can be very large or even infinite and a lot of different strategies to traverse it are imaginable. Which resolution strategy is most

adequate depends on the problem to solve. When all possible solutions should be enumerated, all paths and possible subtypes could be tried in parallel. This kind of fair strategy is save but also very slow. When only a single solution is needed, it might be more convenient to use a depth first strategy based for example on the order in which the features and subtypes have been defined. Such a strategy might be very fast in some cases, but there are also chances that it never terminates - and therefore does not produce any solutions at all.

In our case we are not so much interested in an enumeration of all possible solutions but rather in a single one, which produces a good result from an architectural point of view. For this reason we use an interactive strategy which first generates an initial small generation of random solutions. These then can be evaluated; the best solutions between them can be selected and then used for the generation of a further generation of modified solutions. This process can be iterated until a good solution is found.

For generating the first random generation of solutions, we use a strategy, which we call *random order resolution*. It consists in a depth first search combined with a randomised order for the selection of the features and subtypes applied during the resolution procedure. Therefore, rather than fairly enumerating all solutions, a random solution out of the possible ones is generated. But not in all situations this randomised strategy is adequate: Sometimes values of certain features depend on the values of other features and the order in which they are resolved is important. For these cases it is possible to define an explicit resolution order on all or a subset of the features of a type, which then is respected even by random order resolution.

Some of the solutions generated by random order resolution might be better than others. These solutions can be selected as base for the next generation of solutions, which then are generated as variations or recombinations of the selected ones. By iterating this procedure, the design space can be explored for the most adequate solutions for a given design problem. The variations are generated by slightly modifying the resolution history, i.e. the order of solution steps, which have been used to generate the original solution. Using this modified resolution history for the generation a of new solution results in the latter being similar to the original one, as most of the steps used to generate it are identical.

2.4. Second Iteration: Numerical Constraints - Extending the Simple House Type Constraint System with Size and Location Information

For the exact specification of geometric shapes numerical values are required. When for example assembling a house from a cuboid and a prism, their size and position in relation to the size and position of the house itself

is essential. As numeric constraints are not part of the original definition of TCS as used in computational linguistics, we extended the type system with numerical values and the resolution procedure with a numerical constraint solver.

Let us extend our model of simple houses with the numerical information necessary to generate the actual shapes. The design structure for a simple house currently looks as follows:

$$\begin{bmatrix} house \\ ROOF\ prism \\ BASE\ cuboid \end{bmatrix} \tag{15}$$

It names the parts the example house is assimilated from; but for actually generating it, we need information about the size and location as well:

$$\begin{bmatrix} house & & \\ LOCATION & \langle 0,\ 0,0 \rangle & \\ SIZE & \langle l,\ l,l \rangle & \\ ROOF & \begin{bmatrix} prism & \\ LOCATION & \langle 0,\ 0,l/2 \rangle \\ SIZE & \langle l,\ l,\ l/2 \rangle \end{bmatrix} \\ BASE & \begin{bmatrix} cuboid & \\ LOCATION & \langle 0,\ 0,\ 0 \rangle \\ SIZE & \langle l,\ l,\ l/2 \rangle \end{bmatrix} \end{bmatrix} \tag{16}$$

The vector notation used in this structure is an abbreviation for substructures of type *vector* used to save space:

$$\langle x,\ y,\ z \rangle = \begin{bmatrix} vector \\ X\ x \\ Y\ y \\ Z\ z \end{bmatrix} \tag{17}$$

Looking at this structure we see that the location and size of the house and its parts depend on each other: The location of the house is the same as the location of the base; the location of the roof can be calculated by adding the height of the base to its location; the size of the house finally results from the addition of the sizes of its parts. If we integrate a numerical constraint solver into the resolution procedure, we can formulate these dependencies as part of the constraint for type *house* as shown in Figure 8.

Numerical constraints are resolved whenever enough information is available. In the case of an addition for example, the sum can be calculated, when the two summands are known. And a missing value for a summand can be filled in when the sum and the other summand are determined.

2.5. Third iteration: Design Description Languages and the Generation of the Actual House Shapes

As already explained in the introduction, a design constraint system consists of two components: The design grammar for the generation of the design structures, and the design description language for the interpretation of the design structures and the generation of the actual shapes. Having completed the description of the first component, we now can explain the respective design description language for simple houses.

The design description language for simple houses is straightforward: Only four geometric primitives have to be generated and every primitive has only two parameters: Size and location. For generating the shapes corresponding to the following design structure

$$\left[\begin{array}{ll} \textit{house} & \\ \textit{LOCATION} & \langle 0,\ 0, 0\rangle \\ \textit{SIZE} & \langle \mathrm{I},\ \mathrm{I},\ \mathrm{I}\rangle \\ \textit{ROOF} & \left[\begin{array}{ll} \textit{prism} & \\ \textit{LOCATION} & \langle 0,\ 0, \mathrm{I}/2\rangle \\ \textit{SIZE} & \langle \mathrm{I},\ \mathrm{I},\ \mathrm{I}/2\rangle \end{array}\right] \\ \textit{BASE} & \left[\begin{array}{ll} \textit{cuboid} & \\ \textit{LOCATION} & \langle 0,\ 0,\ 0\rangle \\ \textit{SIZE} & \langle \mathrm{I},\ \mathrm{I},\ \mathrm{I}/2\rangle \end{array}\right] \end{array}\right] \qquad (18)$$

the structure is recursively parsed. When a substructure of type *cuboid, cylinder, prism* or *pyramid* is found, its size and location values are read and the actual shape is generated. In the case of structures with another type

$$\left[\begin{array}{ll} \textit{house} & \\ \text{LOCATION} & \boxed{0}\ \langle \boxed{1},\ \boxed{2},\ \boxed{3}\rangle \\ \text{Size} & \langle \boxed{4},\ \boxed{5},\ \boxed{6}\rangle \\ \text{ROOF} & \left[\begin{array}{ll} \textit{roof} & \\ \text{LOCATION} & \langle \boxed{1},\ \boxed{2},\ \boxed{3}+\boxed{7}\rangle \\ \text{SIZE} & \boxed{8}\ \langle \boxed{4},\ \boxed{5},\ \boxed{7}\ (\boxed{6}\ /\ 2)\rangle \end{array}\right] \\ \text{BASE} & \left[\begin{array}{ll} \textit{base} & \\ \text{LOCATION} & \boxed{0} \\ \text{SIZE} & \boxed{8} \end{array}\right] \end{array}\right]$$

◀ Figure 8. Numerical constraints for type *house*.

nothing is generated and the parsing process continues recursively with the defined feature values.

For the implementation of design description languages we use the free open source 3D content creation suite Blender [11] and generated the geometric shapes as meshes, i.e. the set of vertices and faces defining the shape. The vertices can be calculated easily from the size and location of the respective primitive; the tuples of vertices defining the faces are determined by the morphology of the primitive.

2.6. Forth Iteration: Pagodas and Recursive Type Constraints

The simple houses of our last example are easy to describe: They simply consist of two parts - a base and a roof -, which are stacked to form a house. However, more complex buildings are characterised by larger series of elements like rooms and floors, windows or columns. While the combination of a small and fixed number of parts can be easily described by simple structures, the description of series of objects with single constraints is cumbersome:

$$\begin{aligned} &series \rightarrow series0 \mid seriesI \mid series2 \mid series3 \mid .... \\ &series0, \begin{bmatrix} seriesI \\ OBJI\ obj \end{bmatrix}, \begin{bmatrix} series2 \\ OBJI\ obj \\ OBJ2\ obj \end{bmatrix}, \begin{bmatrix} series3 \\ OBJI\ obj \\ OBJ2\ obj \\ OBJ3\ obj \end{bmatrix}, ... \end{aligned} \tag{19}$$

Fortunately there is a simple way out of this problem: Rather than using a different constraint for every possible number of elements, we can use the difference between adjacent constraints and formulate it as a new constraint, which adds a single feature: Starting from a simple structure with one or even zero elements, we apply this constraint to generate a new structure with one element more. Repeating this procedure, we can generate structures with an arbitrary number of elements. This application of a rule to its own outcome is called *recursion* and it allows the description of an arbitrary number of elements in an elegant and general way.

The following type constraint system for example

$$\begin{aligned} &series \rightarrow empty\text{-}series \mid non\text{-}empty\text{-}series \\ &\begin{bmatrix} non\text{-}empty\text{-}series \\ OBJ\ \ obj \\ OBJS\ series \end{bmatrix} \end{aligned} \tag{20}$$

results in the following series of structures with an arbitrary number of elements:

$$\textit{non-empty-series}, \begin{bmatrix} \textit{non-empty-series} \\ \textit{OBJ} \quad \textit{obj} \\ \textit{OBJS} \quad \textit{empty-series} \end{bmatrix}, \begin{bmatrix} \textit{non-empty-series} \\ \textit{OBJ} \quad \textit{obj} \\ \textit{OBJS} \begin{bmatrix} \textit{non-empty-series} \\ \textit{OBJ} \quad \textit{obj} \\ \textit{OBJS} \quad \textit{empty-series} \end{bmatrix} \end{bmatrix},$$

$$\begin{bmatrix} \textit{non-empty-series} \\ \textit{OBJ} \quad \textit{obj} \\ \textit{OBJS} \begin{bmatrix} \textit{non-empty-series} \\ \textit{OBJ} \quad \textit{obj} \\ \textit{OBJS} \begin{bmatrix} \textit{non-empty-series} \\ \textit{OBJ} \quad \textit{obj} \\ \textit{OBJS} \quad \textit{empty-series} \end{bmatrix} \end{bmatrix} \end{bmatrix}, \ldots \tag{21}$$

Similar to series of elements, pagodas as well can be described by recursion: Starting from a pagoda with only one story, a pagoda with an arbitrary size can be built by repeatedly adding one story after the other. We can reuse our simple houses to model the stories and use recursive type constraints for combining them to pagodas. Let us take the Hōryūji pagoda in Ikaruga, Japan, as model for a new pagoda design constraint system:

Reshaping the simple houses by adapting the value constraints for size and location and renaming the root type to *story*, we can model a story like this:

$$\begin{bmatrix} \textit{story} \\ \textit{LOCATION} \ \langle 0, 0, 0 \rangle \\ \textit{SIZE} \quad \langle 1, 1, 0.6 \rangle \\ \textit{ROOF} \begin{bmatrix} \textit{pyramid} \\ \textit{LOCATION} \ \langle 0, 0, 0.3 \rangle \\ \textit{SIZE} \quad \langle 2.6, 2.6, 0.3 \rangle \end{bmatrix} \\ \textit{BASE} \begin{bmatrix} \textit{cuboid} \\ \textit{LOCATION} \ \langle 0, 0, 0 \rangle \\ \textit{SIZE} \quad \langle 1, 1, 0.3 \rangle \end{bmatrix} \end{bmatrix} \tag{22}$$

A single story can be seen as a simple pagoda with only one story:

$$\begin{array}{l} simple\text{-}pagoda \rightarrow story \\ \begin{bmatrix} simple\text{-}pagoda \\ STORIES\ 1 \end{bmatrix} \end{array} \tag{23}$$

And using a recursive type constraint, we can add a new story to a pagoda of *n* stories in order to generate a pagoda with *n* +1 stories:

$$\begin{bmatrix} complex\text{-}pagoda \\ LOCATION\ \boxed{0}\ \langle \boxed{1}, \boxed{2}, \boxed{3} \rangle \\ SIZE\ \boxed{4}\ \langle \boxed{5}, \boxed{6}, \boxed{7} \rangle \\ STORIES\ \boxed{8}\ (\boxed{9} + 1) \\ TOP \begin{bmatrix} pagoda \\ LOCATION\ \langle \boxed{1}, \boxed{2}, \boxed{3} + 0.7 * \boxed{7} \rangle \\ SIZE\ \boxed{4} \\ STORIES\ \boxed{9} \end{bmatrix} \\ BASE \begin{bmatrix} story \\ LOCATION\ \boxed{0} \\ SIZE\ \boxed{4} \end{bmatrix} \\ CONSTRAINT\ \boxed{8} > 1 \end{bmatrix}$$

(24)

If we adjust the number constraints to make every added story slightly smaller than those already present, we come close to the original shape of the Hōryūji temple:

$$\begin{bmatrix} complex\text{-}pagoda \\ LOCATION\ \boxed{0}\ \langle \boxed{1}, \boxed{2}, \boxed{3} \rangle \\ SIZE\ \boxed{4}\ \langle \boxed{5}, \boxed{6}, \boxed{7} \rangle \\ STORIES\ \boxed{8}\ (\boxed{9} + 1) \\ TOP \begin{bmatrix} pagoda \\ LOCATION\ \langle \boxed{1}, \boxed{2}, \boxed{3} + 0.7 * \boxed{7} \rangle \\ SIZE\ \langle 0.9 * \boxed{5}, 0.9 * \boxed{6}, 0.9 * \boxed{7} \rangle \\ STORIES\ \boxed{9} \end{bmatrix} \\ BASE \begin{bmatrix} story \\ LOCATION\ \boxed{0} \\ SIZE\ \boxed{4} \end{bmatrix} \\ CONSTRAINT\ \boxed{8} > 1 \end{bmatrix}$$

(25)

Integrating the changes, we obtain the following type constraint system for pagodas:

Type Hierarchy :

$$\begin{aligned}
&\textit{pagoda} && \rightarrow \textit{simple - pagoda} \mid \textit{complex - pagoda}\\
&\textit{simple - pagoda} && \rightarrow \textit{story}\\
&\textit{base} && \rightarrow \textit{cuboid}\\
&\textit{roof} && \rightarrow \textit{pyramid}
\end{aligned}$$

Type Constraints:

$$\begin{bmatrix} \textit{simple - pagoda} \\ \textit{STORIES}\ 1 \end{bmatrix}$$

$$\begin{bmatrix}
\textit{complex - pagoda} & \\
\textit{LOCATION} & \boxed{0}\ \langle \boxed{1}, \boxed{2}, \boxed{3} \rangle \\
\textit{SIZE} & \boxed{4}\ \langle \boxed{5}, \boxed{6}, \boxed{7} \rangle \\
\textit{STORIES} & \boxed{8}\ (\boxed{9} + 1) \\
\textit{TOP} & \begin{bmatrix} \textit{pagoda} & \\ \textit{LOCATION} & \langle \boxed{1}, \boxed{2}, \boxed{3} + 0.7 * \boxed{7} \rangle \\ \textit{SIZE} & \langle 0.9 * \boxed{5}, 0.9 * \boxed{6}, 0.9 * \boxed{7} \rangle \\ \textit{STORIES} & \boxed{9} \end{bmatrix} \\
\textit{BASE} & \begin{bmatrix} \textit{story} & \\ \textit{LOCATION} & \boxed{0} \\ \textit{SIZE} & \boxed{4} \end{bmatrix} \\
\textit{CONSTRAINT} & \boxed{8} > 1
\end{bmatrix} \qquad (26)$$

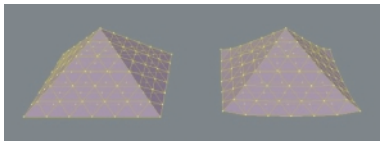
$$\begin{bmatrix}
\textit{story} & \\
\textit{LOCATION} & \boxed{0}\ \langle \boxed{1}, \boxed{2}, \boxed{3} \rangle \\
\textit{SIZE} & \langle \boxed{4}, \boxed{5}, \boxed{6} \rangle \\
\textit{ROOF} & \begin{bmatrix} \textit{roof} & \\ \textit{LOCATION} & \langle \boxed{1}, \boxed{2}, \boxed{3} + \boxed{7} \rangle \\ \textit{SIZE} & \langle 2.6 * \boxed{4}, 2.6 * \boxed{5}, \boxed{7} \rangle \end{bmatrix} \\
\textit{BASE} & \begin{bmatrix} \textit{base} & \\ \textit{LOCATION} & \boxed{0} \\ \textit{SIZE} & \langle \boxed{4}, \boxed{5}, \boxed{7}\ (\boxed{6}/2.0) \rangle \end{bmatrix}
\end{bmatrix}$$

2.7. Fifth Iteration: Pagodas and Distortions

Looking closer at the photograph of the Hōryūji pagoda we notice that its roof shapes are not exactly pyramids: Their corners are slightly lifted up resulting in a gently curved form which gives them a nearly weightless character. While these shapes cannot be generated by the simple volumetric design description language introduced for the generation of simple houses, we can extend it with a geometric transformation which distorts the pyramid shape appropriately for approximating the shape of the original pagoda roofs:

(27)

This distortion can be implemented by subdividing the pyramid mesh and translating vertices appropriately:



(28)

The translation of vertices depend on their distance from the z-axis and can be calculated with a polynomial. For the pagoda roof we used the following formula:

$$d = \sqrt{2x^2 + 2y^2}$$
$$\left(x', y', z'\right) = \left(x, y, z + \frac{1}{8}d^6\right) \quad (29)$$

We could simply add a feature DISTORTION providing the necessary parameters to the pyramid constraint and integrate the polynomial transformation into the code generating the pyramid mesh. But let us use this

► Figure 9. The Hōryūji pagoda in Ikaruga, Japan [12].

opportunity to generalise the design description language in a way that makes it easier to adapt it to other, more complex families of architectural design.

Until now we have added the location and size to the structures of the geometric primitives themselves. But rather than understanding size and location as properties belonging to a single primitive, they can be understood as the result of general transformations that can be applied to geometric primitives and complex shapes as well. The following figure shows the original structure side by side with another one using transformations for scaling, translating and deforming the primitive pyramid shape:

$$\begin{bmatrix} roof & \\ LOCATION & location \\ SIZE & size \\ DISTORTION & distortion \end{bmatrix} \triangleq$$

$$\begin{bmatrix} roof & & & \\ LOCATION & \boxed{0} & location & \\ SIZE & \boxed{1} & size & \\ DISTORTION & \boxed{2} & distortion & \\ SHAPE & \begin{bmatrix} translation & & \\ vector & \boxed{0} & \\ & \begin{bmatrix} scale & \\ SHAPE & factor \ \boxed{1} \\ & \begin{bmatrix} polynomial\text{-}transformation \\ SHAPE & coefficients \ \boxed{2} \\ & SHAPE \ \ pyramid \end{bmatrix} \end{bmatrix} \end{bmatrix} & & \end{bmatrix}$$

(30)

The outermost level of the structure functions as "interface" between the rules of the pagoda design grammar and the nested levels of substructures describing the transformations necessary to generate the appropriate shape.

After integrating the polynomial transformation into the pagoda grammar, the following shape is generated:

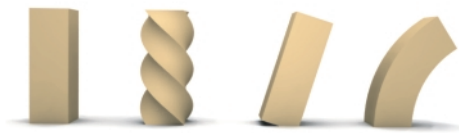
(31)

2.8. Sixth Iteration: Distorted Pagodas and Complex Form

Architecture is bound by the constraints of physical reality: Gravitation and the properties of the used materials define the limits in which architectural design is possible. With the recent development of new materials and construction methods however, the ways in which form and physics go together get more complicated. As a result, the shapes of architecture gain more liberty, and more and more complex shapes and structures become possible. While these advances allow for new ways of architectural expression, they also make the design process much more challenging. For this reason new tools are necessary for making this complexity manageable for the architect and enable her to play and experiment with the new possibilities of complex shapes and structures. While in some cases analogue methods can be used – an example are the hanging force models invented by Antoni Gaudí about 100 years ago for the construction of the Sagrada Família –, in most cases these tools are computer based. As a result the role of the computer in architectural design becomes more and more important, and with it the search for new computer based strategies for the determination of form.

Design constraint systems can be used as tool for experimentation with complex form. We therefore dedicate the rest of this paper to a concise delineation of an approach for the generation of complex and irregular shapes and structures by the application of transformations and distortions to geometric shapes. Of course, this is only one of the possible ways to approach complex form with design constraint systems. Design description languages can also be based on completely different approaches to shape – for example blobs, forms build from splines or skeleton animation techniques as used for example in the free form buildings of Greg Lynn [13,14].

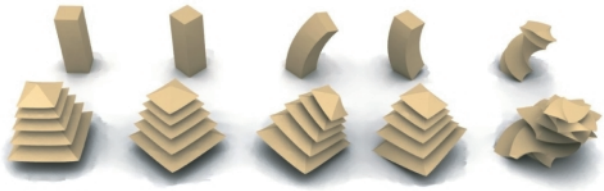
Continuing our method to introduce new aspects of design constraint systems by extending previous models, we use the pagoda grammar as starting point for introducing complex transformations. Let us start by introducing three simple transformations, *rotation*, *twist*, and *bend*:



$$cuboid \begin{bmatrix} twist \\ FACTOR\ \ factor \\ SHAPE\ \ cuboid \end{bmatrix} \begin{bmatrix} rotation\text{-}y \\ ANGLE\ \ angle \\ SHAPE\ \ cuboid \end{bmatrix} \begin{bmatrix} bend \\ RADIUS\ \ radius \\ SHAPE\ \ cuboid \end{bmatrix} \quad (32)$$

The leftmost image shows a simple rectangular block, while the three images on its right show the effect of these transformations on its shape.

Applied to a rectangular block each transformation by itself generates a shape, which is simple and easy to understand. But when interacting with each other, complex forms result. In order to give an example, we extend our design grammar to randomly apply a number of transformations first to the block – to visualize their effect on a simple shape - and later to our pagoda. As the principles of design constraint systems have been explained already, here we show just two results:



(33)

When applying the transformations to the parts of the pagoda, even more complex forms can be generated. And by further customising the design grammar and the design description language, an unrestricted variety of other languages of shape can be defined. There are no limits to the creative use of design constraint systems.

3. CONCLUSION

In the present paper we have introduced design constraint systems, a framework for generative architectural design. First the general assumptions underlying the system have been explained. Then its general organisation into two components – design grammar and design description language – has been outlined. Later, the components themselves and their underlying formalisms have been introduced. For exemplifying the explanations, we started from a model of simple houses, which then was incrementally extended for the generation of pagodas and finally complex shapes and structures.

Apart from models like the pagoda grammar, which generates exactly one model when the parameters size, location, and number of stories are specified unambiguously, design constraint systems in most cases generate a large variety of solutions. We therefore continued by showing how random and evolutionary resolution and the iterative reformulation of the generative rules themselves make solution spaces manageable and help to find good designs in the often large or infinite number of possible ones.

When experimenting with design constraint systems, probably the most interesting aspect is the interaction between randomness and order. Combining shapes and applying transformations in a purely random way rarely results in interesting shapes. Rather it is the hidden logic in which randomness is applied, the balance between order and

chaos, which results in shapes with attractive qualities. The possibility of experimenting with these two opposite aspects of design constraint systems by abstracting rules from existing form, modifying and reapplying them with well regulated randomness, and by starting over with the generated forms and constraints once again until a good design is created is what makes the experimentation with design constraint systems appealing, and a valuable methodology for generative architectural design.

Even if we show just a small number of simple examples, it is easy to see the generality of the introduced approach. By varying the design grammar and the design description language and using random and evolutionary resolution to explore the design space, design constraint systems can be applied to a wide range of architectural styles and design problems.

4. FURTHER INQUIRY

The iterative strategy used in the current paper for explaining design constraint systems not only mirrors the way they were developed, but also indicates how topics for further research should be determined: By applying the system to the modelling of existing styles and types of architecture and by using it for experimenting with new languages of design, the possibilities and necessities to expand and optimise the system become obvious and point to the directions, in which further development has to be invested and the system should be extended. The following list shows some of the most important topics which currently are under development or are planned to be researched in the near future:

- The modelling of other classes of architectural style and the development of the design description languages and design grammars required for this purpose.
- The experimentation with different forms of design description languages and design grammars for the exploration of new ways of architectural expression and problem solving.
- The integration of recombination strategies for designs (cross over) with the goal to generate new designs combining the characteristics of different existing ones and as a new evolutionary search strategy for design spaces. The current search strategy is based on random variation (mutation) only.
- Ways to freeze certain parts of the design structure for experimentation with partial aspects of designs, for example the compositional structure the colour or height etc.
- The integration of automatic evaluation procedures, for example by calculating the structural fitness [4], the lightening conditions etc. for experimenting with non-interactive modes of evolutionary design.

- The extension of the current approach, which focuses on architectural morphology only, for the integration of other aspects of architecture like functional criteria etc. [15].
- Finally the optimisation of the system itself. Some parts of the current system are planned to be reimplemented for better performance. The resolution state for example currently is copied during each resolution step. A lazy strategy, which only represents the changes between different resolution states, would result in a more efficient system.

Design constraint systems propose a general paradigm for generative design. A large number of different possibilities to extend them therefore exist - and new ideas are always welcome. The given list of directions for further research therefore is only meant as a source of inspiration. New directions for development should be a result of the needs and ideas ensuing from the continuous application of DCS to design problems and the curiosity and creativity of the researchers involved.

Acknowledgement

We would like to thank Professor Fujii and Professor Imai of the University of Tokyo for their support of the introduced research project. Thanks to countless discussions and suggestions, the members of the Fujii/Imai laboratory as well played an important role for the development of the described system and approach to architectural design. In especially we would like to express our gratitude to Yukie Ogoma, Kenichiro Hashimoto, Dr. Glen Wash, Dr. Esteban Beita and Dr. Wanwen Huang for there ideas and cooperation.

References

1. Woodbury, R., *Elements of Parametric Design*, Routledge, London, 2010.
2. Stiny, G., *Shape: Talking about Seeing and Doing*, MIT Press, Cambridge, Massachusetts, 2006.
3. Sasaki, M., *Morphogenesis of Flux Structure*, AA Publications, London, 2007.
4. Preisinger, C., Linking Structure and Parametric Geometry. *Architectural Design (AD)*, 83: 110-113, John Wiley & Sons, London, UK, 2013.
5. Carpenter, B., *The Logic of Typed Feature Structures*, Cambridge University Press, New York, 1992.
6. Pollard, C. and Sag, I. A., *Information-based Syntax and Semantics*, Vol. 1, Number 13 in Lecture Notes, CSLI Publications, Stanford University, University of Chicago Press, Chicago, 1987.
7. Pollard, C. and Sag, I. A., *Head-Driven Phrase Structure Grammar*, University of Chicago Press, Chicago, 1994.
8. Bollmann, D., A Generative Approach to Architectural Form using Design Constraint Systems - Case Studies in Vernacular Compounds in Burkina Faso, PhD thesis, The University of Tokyo, Department of Architecture, Tokyo, Japan, 2012.

9. De Jong, K. A., *Evolutionary Computation, A Unified Approach*, MIT Press, Cambridge, Massachusetts, 2006.
10. Aït-Kaci, H., *A lattice theoretic approach to computation based on a calculus of partially ordered type structures (property inheritance, semantic nets, graph unification)*, PhD Thesis, Philadelphia, PA, USA, 1984.
11. See the Blender homepage at http://www.blender.org/.
12. This photograph has been taken from the Wikipedia and is licenced under the terms of the GNU Free Documentation License, see http://en.wikipedia.org/wiki/File:Horyu-ji06s3200.jpg.
13. Lynn, G., Animate Form, Princeton Architectural Press, New York, USA, 1999.
14. Lynn, G. and Rappolt, M., Eds., Greg Lynn Form, Rizzoli, New York, USA, 2008.
15. Woodbury, R., Burrow, A., Datta, S. and Chang, T.-W., Typed feature structures and design space exploration. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 13:287-302, Cambridge University Press, New York, USA, 1999.

Dietrich Bollmann, Ph.D.
Dept. of Architecture, The University of Tokyo, Japan

dietrich@formgames.org

Alvaro Bonfiglio, Ph.D.
School of Architecture Universidad de la Republica
Uruguay

abonfigl@adinet.com.uy