Computer Assisted Mechanics and Engineering Sciences, 10: 93–109, 2003.

Graph transformations in architectural design

Janusz Szuba and Adam Borkowski
Institute of Fundamental Technological Research, Polish Academy of Sciences
ul. Świętokrzyska 21, 00-049 Warsaw, Poland

(Received January 10, 2002)

This paper deals with computer-aided design of layouts of buildings. The methodology based upon graph grammars and graph transformations allows the designer to distract itself from details and to consider the functionality of the designed object, the constraints and the requirements to be met and the possible ways of selecting optimum alternatives. The specification of building is made in the UML with the aid of the FUJABA system. After this has been accomplished, a proper graph grammar is generated automatically. Such a grammar defines a class of objects that fulfill prescribed requirements and deliver required functionality. The user of the proposed system can browse members of that class, i.e. compare alternative plausible designs, using any commercially available visualization tool.

1. Introduction

Pioneered by N. Chomsky [8] the linguistic approach to world modeling found applications in many areas. The core idea in this methodology is to treat certain primitives as letters of an alphabet and to interpret more complex objects and assemblies as words or sentences of a language based upon the alphabet. Rules governing generation of words and sentences define a grammar of the concerned language. In terms of the world modeling such a grammar generates a class of objects that are considered plausible. Thus, grammars provide very natural knowledge representation formalism for computer-based tools that should aid the design.

Since G. Stiny [22] has developed the shape grammars many researchers showed how such grammars allow the architect to capture essential features of a certain style of the building (e.g. Victorian houses or Roman villas). However, the primitives of shape grammars are purely geometrical which restricts their descriptive power. Substantial progress was achieved after the graph grammars were introduced and developed (compare, e.g. [20]). Graphs are capable to bear much more information than linear strings or shapes. Hence, their applicability for CAD-systems was immediately appreciated [12].

A special form of graph-based representation has been developed by E. Grabska [13, 14]. This formalism distinguishes the composition graphs (CP-graphs) that describe the structure of the object from the realization schemes describing the visualization. In 1996–98 Grabska's model served as the basic knowledge representation scheme in the research project [5] aimed at developing intelligent design-assisting tools for engineering. The results of that project were reported at the conferences in Stanford [15], Ascona [4] and Wierzba [3].

It turned out that by introducing an additional functionality graph into the original Grabska's model one can conveniently reason about conceptual solutions for the designed object. The functionality analysis as the starting point of the conceptual design has been proposed by several researchers (compare, e.g. [6, 9]). Such methodology allows the designer to distract himself from details and to consider the functionality of the designed object, the constraints and the requirements to be met and the possible ways of selecting optimum alternatives.

In the sequel we present results obtained in co-operation with E. Grabska, M. Nagl and A. Schürr under a joint research project [7] The aim of this project is to develop prototype software that will assist an architect in the design of the layout of building. Contrary to conventional expert systems proposed previously, like [11], our system can be seen as a conceptual preprocessor for an architecture-oriented CAD-tool. It allows the user to specify functional requirements for a single-family house in terms of graphs, generates a proper graph grammar and translates the result into the input file for the CAD-system ArchiCAD [1]. The architect obtains a draft layout of the house that can be visualized and presented to the investor.

Fast prototyping is important in many areas and architecture is no exception. It enables the designer to present the draft of the design to the client in short time and to achieve approval or disapproval of the presented proposition. In this way the designer knows the intention of the client. If the client accepts the general conception of the design, the architect can start working on details.

Our system is generative: it does not produce a single layout but an entire family of plausible layouts described by the graph grammar. In order to achieve that we apply the Unified Modeling Language (UML) [2] for the specification purposes and we take advantage of the FUJABA [10] — a convenient Java-based graph editor. It is our intention to replace in the future FUJABA by the more powerful generative system PROGRESS [21] developed at the RWTH, Aachen. The results described in Sections 2 to 4 can be viewed as the further development of the research reported previously in [16, 23, 24].

2. Graphs in design process

2.1. General considerations

In this section we show how graph transformations can be used in the conceptual phase of architectural design. This is the phase when the architect knowing functional requirements and constraints from the conversation with the client sketches the layout of the building (compare the part of Fig. 1 above the horizontal line). These sketches are further presented to the client and discussed with him. Finally, one of the created sketches becomes the basis for the complete design (compare, e.g. [18] or [19]).

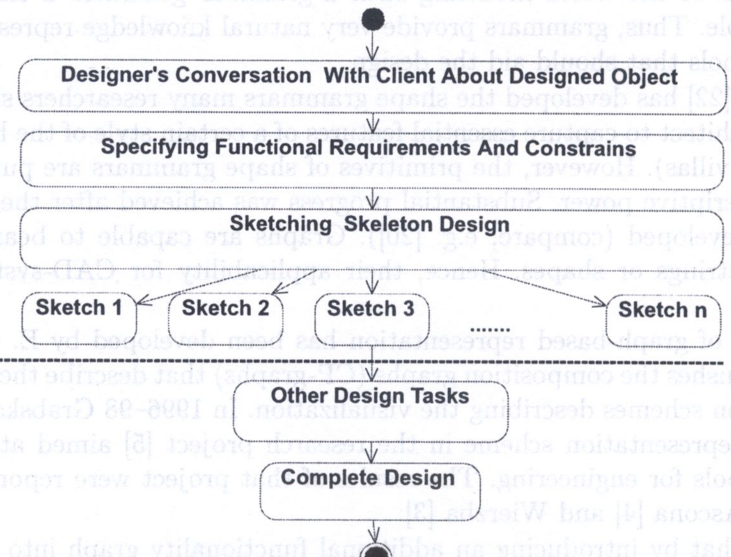

Designer's Conversation With Client About Designed Object
Specifying Functional Requirements And Constrains
Sketching Skeleton Design
Sketch 1
Sketch 2
Sketch 3
....
Sketch n
Other Design Tasks
Complete Design

Fig. 1. Process of designing – traditional approach

The scheme given in Fig. 1 is deliberately simplified. The real design process includes repetitive loops, abandoning partial solutions and backtracking. The aim of this scheme is merely to emphasize the role of sketching conceptual alternatives at the initial stage.

Our proposal amounts to giving the designer a computer-based tool that allows him to reason about functional requirements and to transform them into the structural scheme of the designed object. In order to achieve that, we distinguish four phases of the design process:

1. Specifying functional requirements for the designed object.
2. Transforming them into the structure of the object.
3. Visualising the object.
4. Working out the detailed design.

We propose the initial two steps to be graph-based. Firstly, the designer takes the list of required functions and constructs *a functionality graph.* The nodes of this graph correspond to the functions that the considered artefact has to fulfill. The edges connecting the nodes depict functional relationships.

In the second phase the functionality graph is mapped into *a structural graph* of the object. The nodes of the structural graph correspond to the components of the object; the edges represent relations between the components. Thus, the structural graph describes *a physical decomposition* of the artefact. By assigning the functions to the components one aims at satisfying all the functional requirements by the object viewed as an assembly of its components. The transition from the functionality graph to the structural graph is neither unique nor straightforward. Hence, the designer needs usually several iterative loops over the steps 1, 2 before the satisfactory solution is found (Fig. 2).

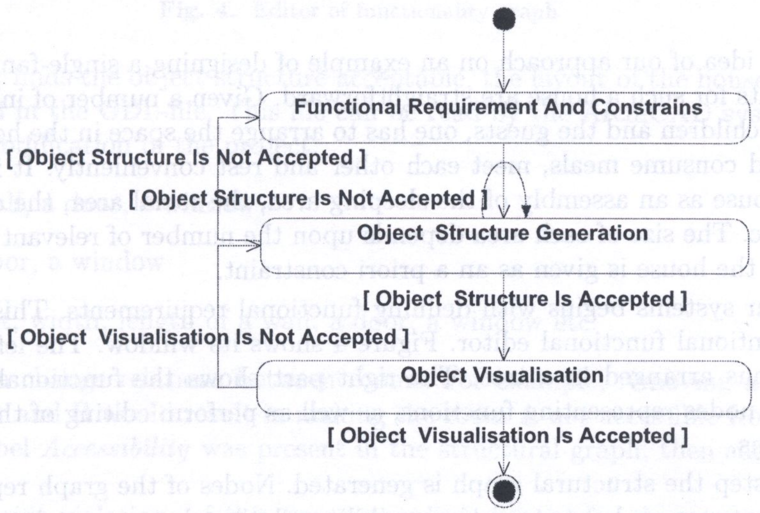

Functional Requirement And Constrains
[Object Structure Is Not Accepted]
[Object Structure Is Not Accepted]
Object Structure Generation
[Object Structure Is Accepted]
[Object Visualisation Is Not Accepted]
Object Visualisation
[Object Visualisation Is Accepted]

Fig. 2. Designing process with graph transformations

The step 3 — the visualization of the object based upon the structural graph — is performed automatically. Our system generates the floor layout of the building in the format accepted by the commercially available CAD-system ArchiCAD [1]. This system allows the user to visualize the building in 2D or 3D mode, to make any desired cross-section of the building and to assign its components the elements of the library of standard units, like the doors, the windows, etc.

Usually the visualization reveals several drawbacks of the designed object. Such errors are removed by revisiting the steps 1 and 2. After several loops the final solution is found and the project enters the step 4. The detailed design is accomplished in usual manner by means of the ArchiCAD (Fig. 3).

The advantages of using graph-based representation during the steps 1, 2 seem to be obvious. The designer is encouraged to think firstly in the abstract terms of functions and their relations.

After the functionality graph is established, the designer tries to decompose the object into physical units assigning particular functions to them. The entire process is performed graphically by means of the editor that allows the designer to manipulate graphs conveniently.

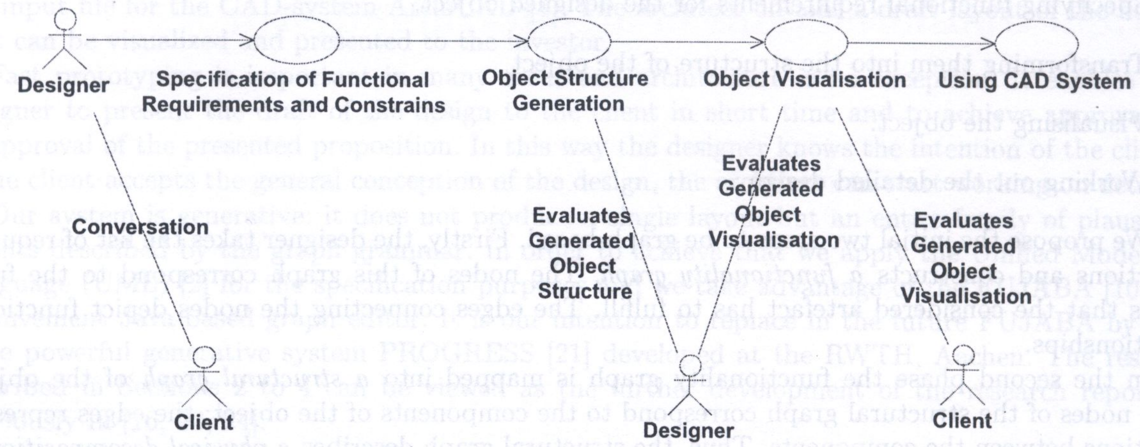

Designer
Specification of Functional Requirements and Constrains
Object Structure Generation
Object Visualisation
Using CAD System
Conversation
Evaluates Generated Object Structure
Evaluates Generated Object Visualisation
Evaluates Generated Object Visualisation
Client
Designer
Client

Fig. 3. Designing process with graph transformations

2.2. Case study

Let us clarify the idea of our approach on an example of designing a single-family house. The functional requirements for such a house are straightforward. Given a number of inhabitants, consisting of the adults, the children and the guests, one has to arrange the space in the house so that they can sleep, prepare and consume meals, meet each other and rest conveniently. It is natural, therefore, to consider the house as an assembly of the sleeping area, the social area, the communication area, the guest area, etc. The size of each area depends upon the number of relevant class of inhabitants. The total area of the house is given as an a priori constraint.

The user of our systems begins with defining functional requirements. This is accomplished by means of a conventional functional editor. Figure 4 shows its window. The left part of it contains the list of functions arranged in a tree. The right part shows the functionality graph. The user can add or delete nodes representing functions, as well as perform editing of the edges representing functional relations.

In the second step the structural graph is generated. Nodes of the graph represent the rooms of the house; edges between nodes represent accessibility relations between rooms. After the structural graph has been generated, the designer evaluates it. Note that at this stage the level of abstraction remains high: the geometry is still irrelevant, what matters is the assignment of functions to particular rooms.

If the evaluation falls negative, the designer can modify the structure. In terms of graphs this means adding or deleting a node, adding or deleting an edge, merging two nodes into one, splitting a node into two, changing the type of node or changing the label of edge. Given the editor the architect is not bothered by the technicalities of the graph theory. Adding (deleting) the node is for him adding (deleting) the room in the house, adding (deleting) the edge is adding (deleting) the passage from one room to another, merging two nodes is merging two rooms into one (e.g., joining bathroom and WC), splitting two nodes is making two rooms out of one (e.g., dividing the hall into two parts), changing the node type is changing the function of the room (e.g., changing a dining room into a sleeping room), changing the label of edge is changing the relation between rooms. Architects are trained in visual reasoning. Therefore, they find this tool quite intuitive.

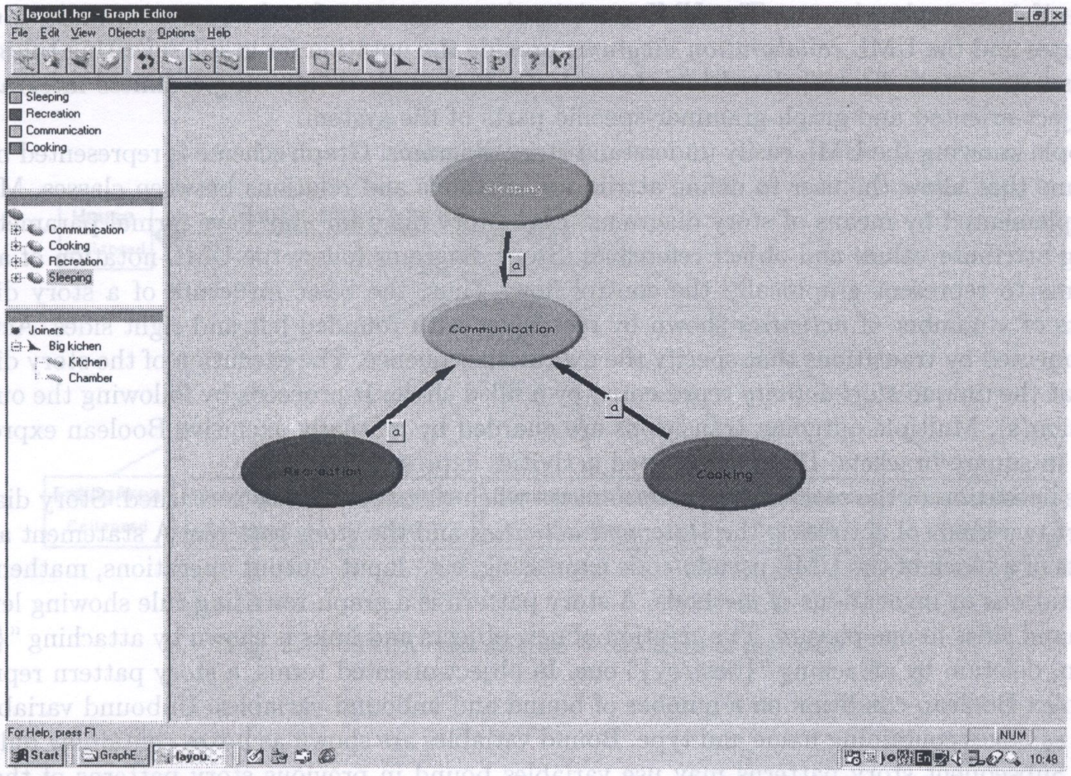


Fig. 4. Editor of functionality graph

After the designer finds the object structure acceptable, the layout of the house is automatically generated and stored in the GDL-file. This file can be read by the ArchiCAD system and the user can perform usual modification of the project:

1. Add (delete) a wall, a door, a window
2. Move a wall, a door, a window
3. Change the height, width, length of a wall, a door, a window etc.

These operations can change relations between rooms. For example, removing a door in the wall between the rooms A and B would result in making the room A not accessible from the room B. If the edge with the label *Accessibility* was present in the structural graph, then such modification is not allowed.

At present constraint violations of this type are only indicated for the user and the corrective action must be performed manually. We intend to develop in the future an advanced version of the system that will support the backtracking from the ArchiCAD to higher levels of abstraction, namely, to the functionality and structural graphs. This is a challenging task since a supervisory control layer must be developed. The aim of this layer will be to co-ordinate bi-directional information flow between the graph-oriented part of the system and the visualization oriented CAD module.

3. Generating Floor Layout

3.1. Software tools and languages

In our research we use FUJABA (**F**rom **U**ML to **J**ava **a**nd **B**ack **A**gain) — a generative system developed and implemented in Java at the Paderborn University (Germany) [10]. FUJABA contains graph grammar language called *story diagrams.* This language uses the UML [2] *class diagrams* for specifying graph schemes. The UML *activity diagrams* serve for the representation of control structures and the UML *collaboration diagrams* provide the notation for graph rewriting rules. Story diagrams are internally translated into Java classes and methods allowing seamless integration of the object-oriented and graph-grammar-specific parts of the system.

People knowing the UML easily understand story diagrams. Graph scheme is represented by class diagrams that allow the user to define attributes, methods and relations between classes. Methods are implemented by means of story diagrams. Each story diagram may have formal parameters for passing attribute values and object references. Story diagrams follow the UML-notation of activity diagrams to represent graphically the control flow. Thus, the basic structure of a story diagram consists of a number of *activities* shown by rectangles with rounded left and right sides. Activities are connected by transitions that specify the execution sequence. The execution of the story diagram starts at the unique *start activity* represented by a filled circle. It proceeds by following the outgoing transition(s). Multiple outgoing transitions are guarded by mutually exclusive Boolean expressions shown in square brackets. Diamond-shaped activities express branching.

The execution of the story diagram terminates when the *stop activity* is reached. Story diagrams support two kinds of activities: the *statement activities* and the *story patterns.* A statement activity consists of a block of the UML pseudo-code expressing, e.g., input–output operations, mathematical computations or invocations of methods. A story pattern is a graph-rewriting rule showing left- and right-hand sides in one picture. The creation of new objects and links is shown by attaching "{new}" caption, deletion by attaching "{destroy}" one. In object-oriented terms, a story pattern represents a complex Boolean condition on a number of bound and unbound variables. Unbound variables are shown as boxes containing name and type. Bound variables are shown as boxes containing only their name. Subsequent story patterns may use variables bound in previous story patterns of the same story diagram. A link in a story pattern represents the Boolean condition that the objects matched by the corresponding variables are connected by such a link. Generally, a story pattern is executed by binding all its unbound variables to objects such that the represented condition evaluates to true. If this is possible, the specified modifications are performed and the story pattern succeeds, otherwise it fails.

3.2. Class diagrams

The first step in describing a specification is constructing a class diagram where the objects that will be transformed and the relations between them are defined. The class diagrams relevant for the specification of single-family house are shown in Figs. 5, 6 and 7.

The most important class in this specification is the class *House* that represents the entire building. The class *Area* is the base class for the subclasses *SleepingArea*, *CommunicationArea*, *RelaxationArea*, *EatingArea*, *GuestArea*, *CleaningArea*. These subclasses bear the main functions of the house. The class *Room* is the base class for subclasses *BedRoom*, *BathRoom*, *GuestRoom*, *WC*, *Kitchen*, *DinningRoom*, *Hall*, *LivingRoom* that represent physical decomposition of the building. Finally, the class *User* is the base class for subclasses *Adult*, *Child*, *Guest* that represent types of inhabitants.

The following relations are defined between classes:

1. *HouseContainsArea* — between *House* and *Area*.

2. *AreaContainsRoom* — between *Area* and *Room*.

3. *Uses* — between *User* and *Room*.

4. *AreaAccessibility* — between *Area_1* and *Area_2*.

5. *RoomAccessibility* between *Room_1* and *Room_2*

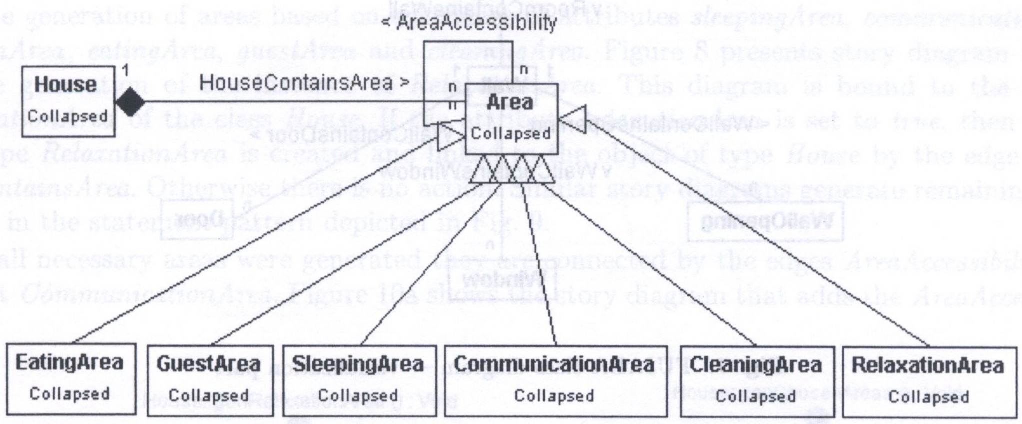


Fig. 5. FUJABA class diagram — the areas of the house

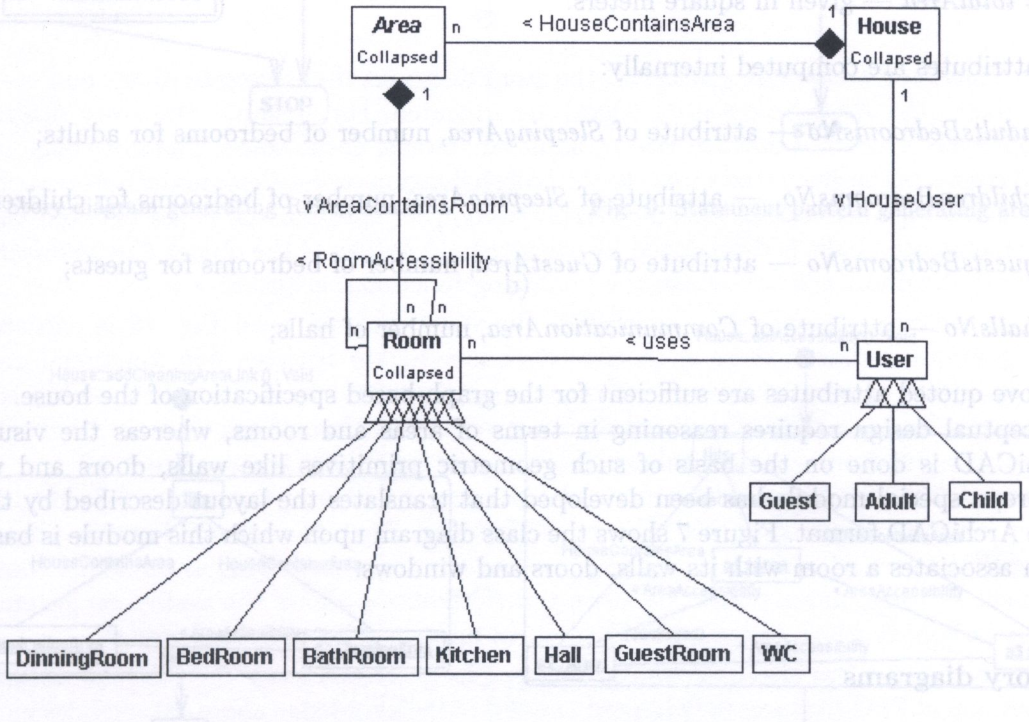


Fig. 6. FUJABA class diagram — rooms and inhabitants

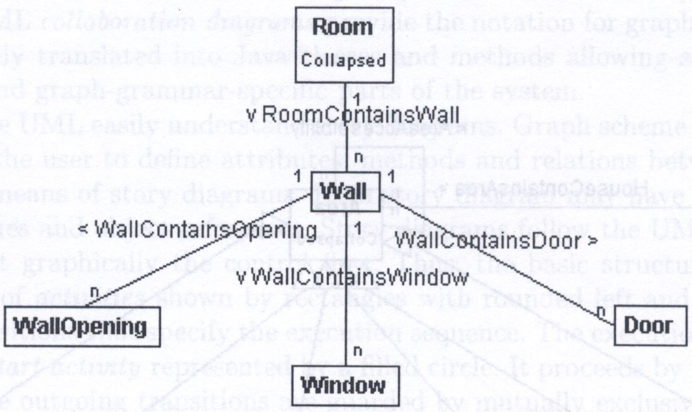


Fig. 7. FUJABA class diagram — visualization part

The following attributes of the object *House* obtain their values after interrogating the client:

1. *int adultsNo* — number of adult inhabitants;

2. *int childrenNo* — number of children;

3. *int guestsNo* — number of guests;

4. *boolean sleepingArea, communicationArea, relaxationArea, eatingArea, guestArea, cleaningArea* — flags of functions;

5. *float totalArea* — given in square meters.

Other attributes are computed internally:

1. *int adultsBedroomsNo* — attribute of *SleepingArea*, number of bedrooms for adults;

2. *int childrenBedroomsNo* — attribute of *SleepingArea*, number of bedrooms for children;

3. *int guestsBedroomsNo* — attribute of *GuestArea*, number of bedrooms for guests;

4. *int hallsNo* — attribute of *CommunicationArea*, number of halls;

The above quoted attributes are sufficient for the graph-based specification of the house.

Conceptual design requires reasoning in terms of areas and rooms, whereas the visualization in ArchiCAD is done on the basis of such geometric primitives like walls, doors and windows. Therefore, a special module has been developed that translates the layout described by the graph into the ArchiCAD format. Figure 7 shows the class diagram upon which this module is based. This diagram associates a room with its walls, doors and windows.

3.3. Story diagrams

As already mentioned in Section 2, story diagrams describe graph transformations and are related to methods of classes specified in class diagrams. In this Section we present graph transformation part of our FUJABA specification. In the sequel we refer to this specification as FLG standing for the Floor Layout Generation.

3.3.1. *Rules on areas*

The execution of the FLG specification begins with the creation of the instance of class *House*. The attributes of this object are initiated to the values mentioned in the previous Section. The next step is the generation of areas based on the values of attributes *sleepingArea*, *communicationArea*, *relaxationArea*, *eatingArea*, *guestArea* and *cleaningArea*. Figure 8 presents story diagram that realizes the generation of the instance of *RelaxationArea*. This diagram is bound to the method *genRelaxationArea* of the class *House*. If the attribute *relaxationArea* is set to *true*, then the object of type *RelaxationArea* is created and linked to the object of type *House* by the edge labeled *HouseContainsArea*. Otherwise there is no action. Similar story diagrams generate remaining areas, as shown in the statement pattern depicted in Fig. 9.

After all necessary areas were generated they are connected by the edges *AreaAccessibility* with the object *CommunicationArea*. Figure 10a shows the story diagram that adds the *AreaAccessibility*

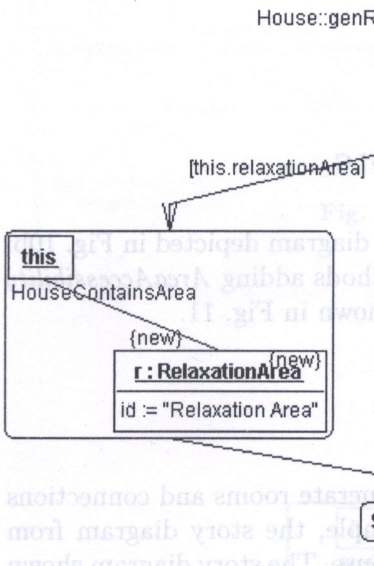
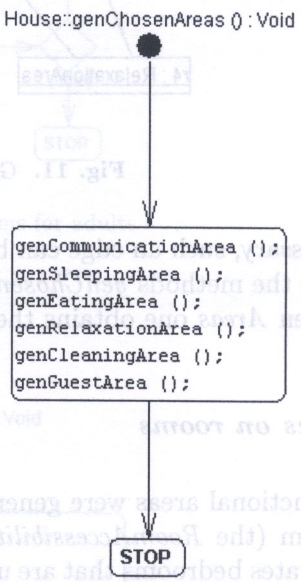


Fig. 8. Story diagram generating RelaxationArea



```
genCommunicationArea ();
genSleepingArea ();
genEatingArea ();
genRelaxationArea ();
genCleaningArea ();
genGuestArea ();
```

Fig. 9. Statement pattern generating areas

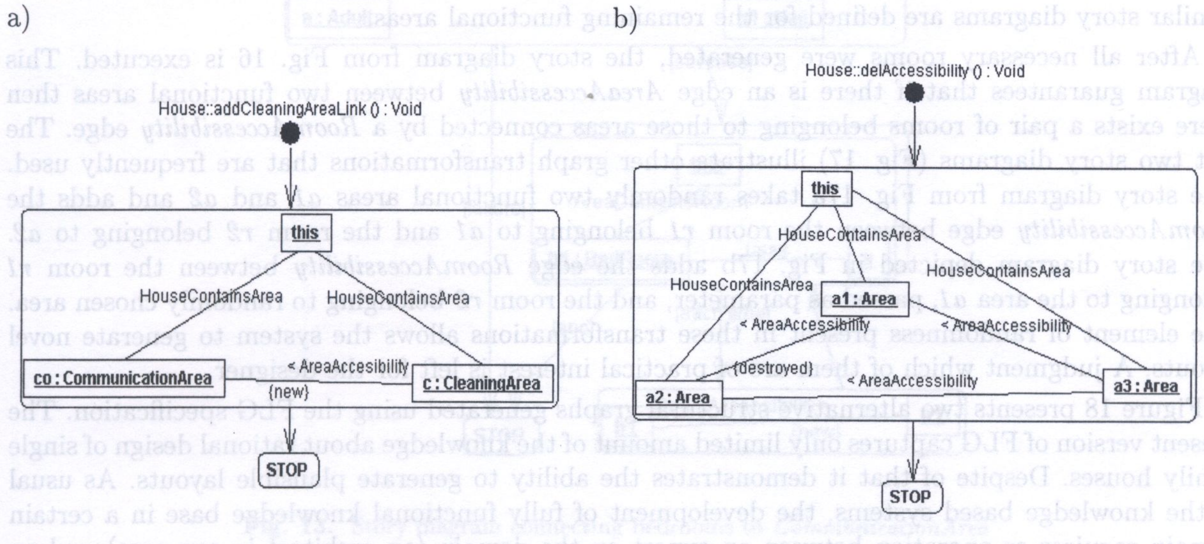


Fig. 10. Story diagrams related to AreaAccessibility edge: a) adding edge; b) removing edge

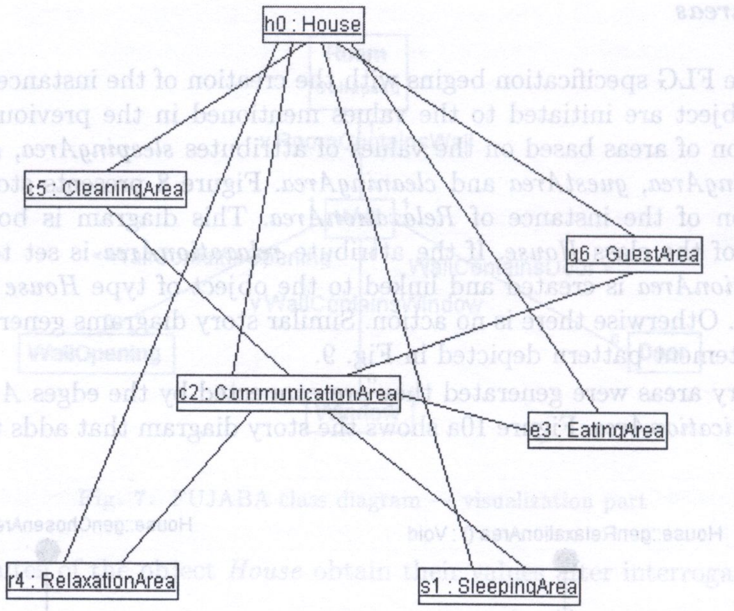


Fig. 11. Graph of house generated by means of FLG

edge. If necessary, such an edge can be removed by means of the story diagram depicted in Fig. 10b. After calling the methods *genChosenAreas* and *genAreaRelations* methods adding *AreaAccessibility* edges between *Areas* one obtains the structural graph of the house shown in Fig. 11.

3.3.2. Rules on rooms

After the functional areas were generated, the story diagrams that generate rooms and connections between them (the *RoomAccessibility* edges) are executed. For example, the story diagram from Fig. 12 generates bedrooms that are used by adult inhabitants of the house. The story diagram shown in Fig. 13 connects the generated rooms to the halls belonging to the *CommunicationArea*. The story diagrams from Figs. 14 and 15 generate, respectively, the halls and the edges between them. The number of generated halls is determined by the attribute *hallNo* of the object *CommunicationArea*. Similar story diagrams are defined for the remaining functional areas.

After all necessary rooms were generated, the story diagram from Fig. 16 is executed. This diagram guarantees that if there is an edge *AreaAccessibility* between two functional areas then there exists a pair of rooms belonging to those areas connected by a *RoomAccessibility* edge. The last two story diagrams (Fig. 17) illustrate other graph transformations that are frequently used. The story diagram from Fig. 17a takes randomly two functional areas *a1* and *a2* and adds the *RoomAccessibility* edge between the room *r1* belonging to *a1* and the room *r2* belonging to *a2*. The story diagram depicted in Fig. 17b adds the edge *RoomAccessibility* between the room *r1* belonging to the area *a1*, passed as parameter, and the room *r2* belonging to randomly chosen area. The element of randomness present in these transformations allows the system to generate novel layouts. A judgment which of them are of practical interest is left for the designer.

Figure 18 presents two alternative structural graphs generated using the FLG specification. The present version of FLG captures only limited amount of the knowledge about rational design of single family houses. Despite of that it demonstrates the ability to generate plausible layouts. As usual in the knowledge based systems, the development of fully functional knowledge base in a certain domain requires co-operation between an expert on the domain (an architect in our case) and an expert on graph transformations that would write down necessary formal structures.

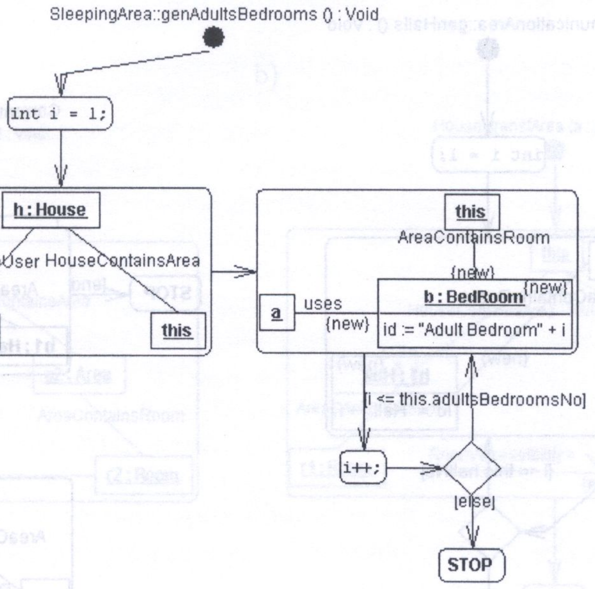


Fig. 12. Story diagram generating bedrooms for adults

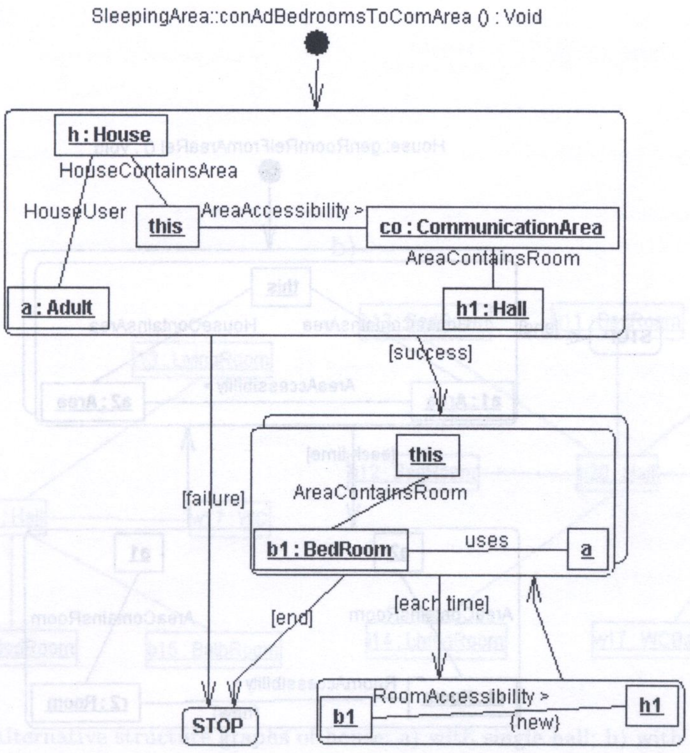


Fig. 13. Story diagram connecting bedrooms to *CommunicationArea*

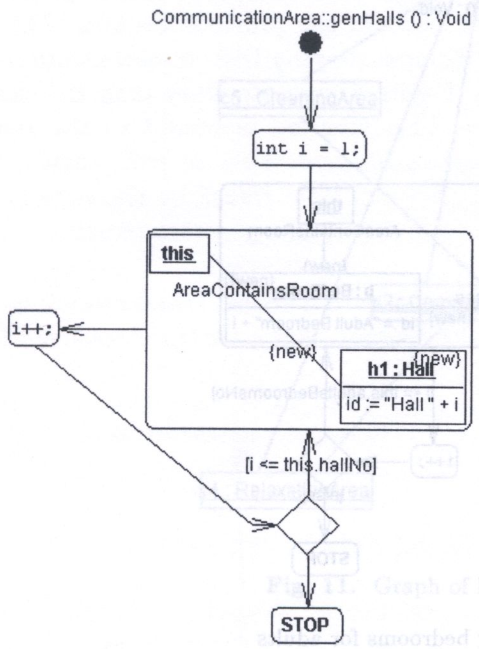


Fig. 14. Story diagram generating halls

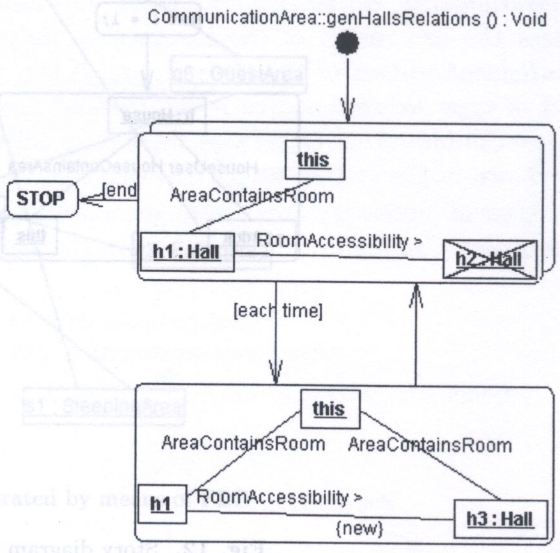


Fig. 15. Story diagram connecting halls

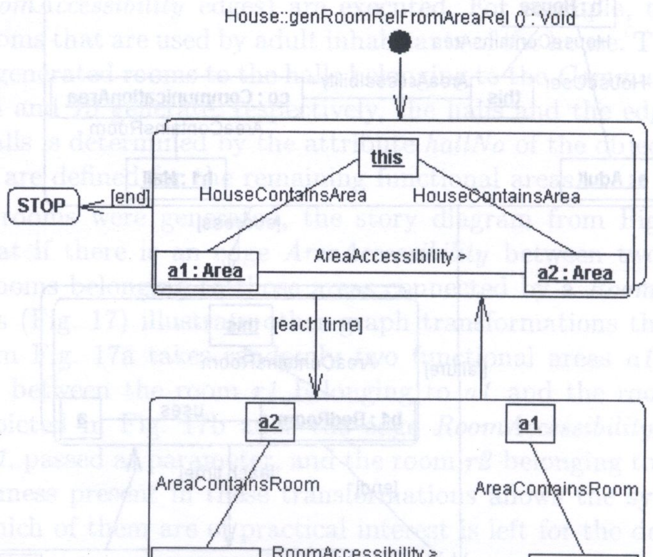


Fig. 16. Story diagram generating room relations from area relations

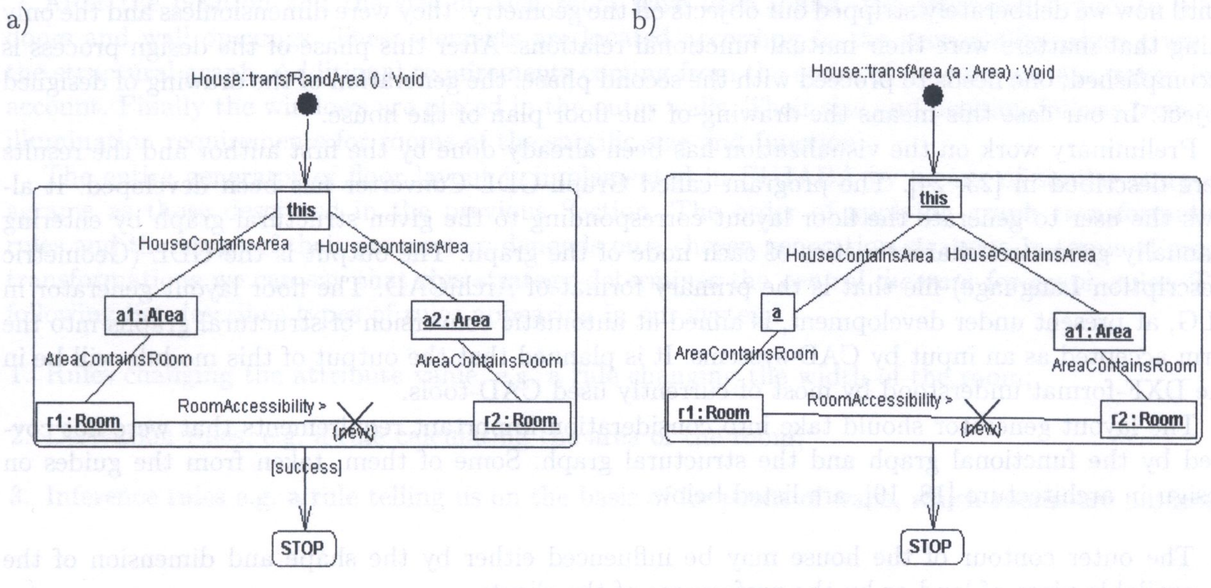


Fig. 17. Story diagram adding *RoomAccessibility* edge: a) between two randomly chosen areas; b) between area given in the parameter of the story diagram and randomly chosen area

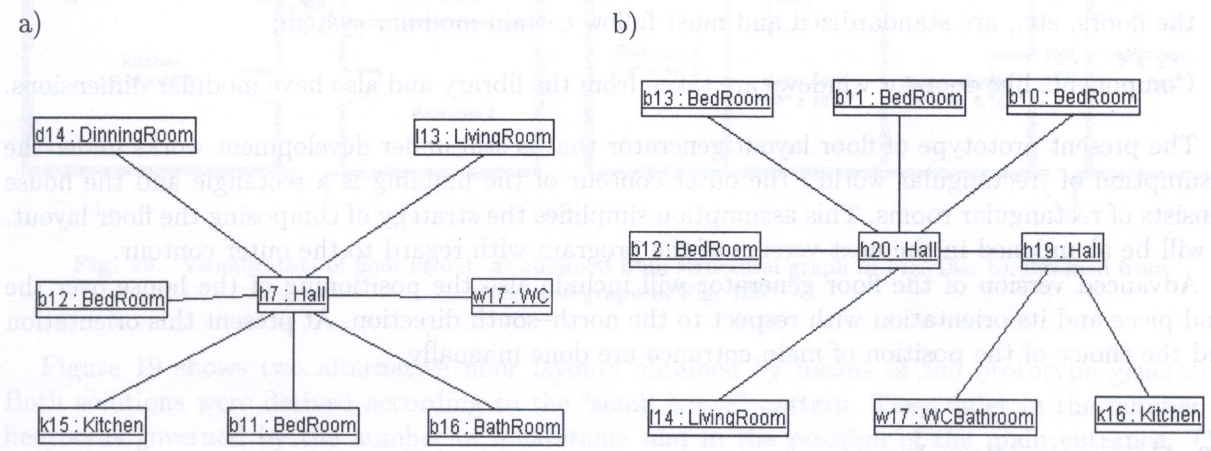


Fig. 18. Alternative structure graphs of house: a) with single hall; b) with two halls

4. Implementation issues

4.1. Functionality of the object visualization module

Until now we deliberately stripped our objects off the geometry: they were dimensionless and the only thing that matters were their mutual functional relations. After this phase of the design process is accomplished, one needs to proceed with the second phase: the generation of the drawing of designed object. In our case this means the drawing of the floor plan of the house.

Preliminary work on the visualization has been already done by the first author and the results were described in [23, 24]. The program called Graph-GDL Converter has been developed. It allows the user to generate the floor layout corresponding to the given structural graph by entering manually geometric characteristics of each node of the graph. The output is the GDL (Geometric Description Language) file that is the primary format of ArchiCAD. The floor layout generator in FLG, at present under development, is aimed at automatic conversion of structural graphs into the form accepted as an input by CAD-systems. It is planned that the output of this module will be in the DXF-format understood by most of currently used CAD-tools.

The layout generator should take into consideration important requirements that were not covered by the functional graph and the structural graph. Some of them, taken from the guides on design in architecture [18, 19], are listed below:

1. The outer contour of the house may be influenced either by the shape and dimension of the available piece of land or by the preferences of the client;

2. The orientation of the house with respect to the north–south axis must be taken into account when placing the rooms (e.g., a living room should be located in the south, whereas a kitchen in the north);

3. There are minimal values of areas for particular types of rooms (e.g., the area of living room should not be less than 18 m^2);

4. Most rooms should be kept as close as possible to square shape (exceptions are communication areas and special purpose units);

5. Main dimensions of the building, like the distances between load carrying walls, the height of the floors, etc., are standardized and must follow certain modular system;

6. Components like doors or windows are taken from the library and also have modular dimensions.

The present prototype of floor layout generator that is still under development works under the assumption of "rectangular world": the outer contour of the building is a rectangle and the house consists of rectangular rooms. This assumption simplifies the strategy of composing the floor layout. It will be abandoned in the next version of the program with regard to the outer contour.

Advanced version of the floor generator will include also the positioning of the house over the land piece and its orientation with respect to the north–south direction. At present this orientation and the choice of the position of main entrance are done manually.

4.2. Generating floor layout

The floor layout generator scans the structural graph of the house node by node and creates "embryos" of rooms: the objects that have already walls but are of square shape with minimum allowable area. Such embryos are placed inside the preliminary contour of the floor according to heuristic rules. Then they are allowed to expand until there is no free space between adjacent rooms. The preliminary outer contour is allowed to adjust itself in order to accommodate all necessary rooms.

The initial placement of room embryos follows three predefined patterns. The choice of pattern depends upon the global area of the house requested by the client. The rooms in small houses are placed around a corridor leading from the main entrance. For medium size houses this corridor is replaced by a hall and for big houses — by an internal garden or atrium.

After the position and the size of each room have been found, the generator begins to place doors and wall openings. These elements are located according to the accessibility edges given in the structural graph. Additional requirements coming from the codes of practice are also taken into account. Finally the windows are placed in the outer walls. Their size and position follows from the illumination requirements for rooms of the specific size and function.

The entire generator of floor layout is implemented in FUJABA by means of similar story diagrams as those described in the previous Section. The order of applying graph transformation rules and the usage of their parameters depends on a chosen generation strategy. In terms of graph transformations we can say that this strategy determines the *control diagram* for graph rules. The following list describes types of rules appearing in our system:

1. Rules changing the attribute value, e.g. a rule changing the width of the room;

2. Derivation rules, e.g. a rule calculating the area of the room;

3. Inference rules e.g. a rule telling us on the basis of locations of walls, which rooms are adjacent.

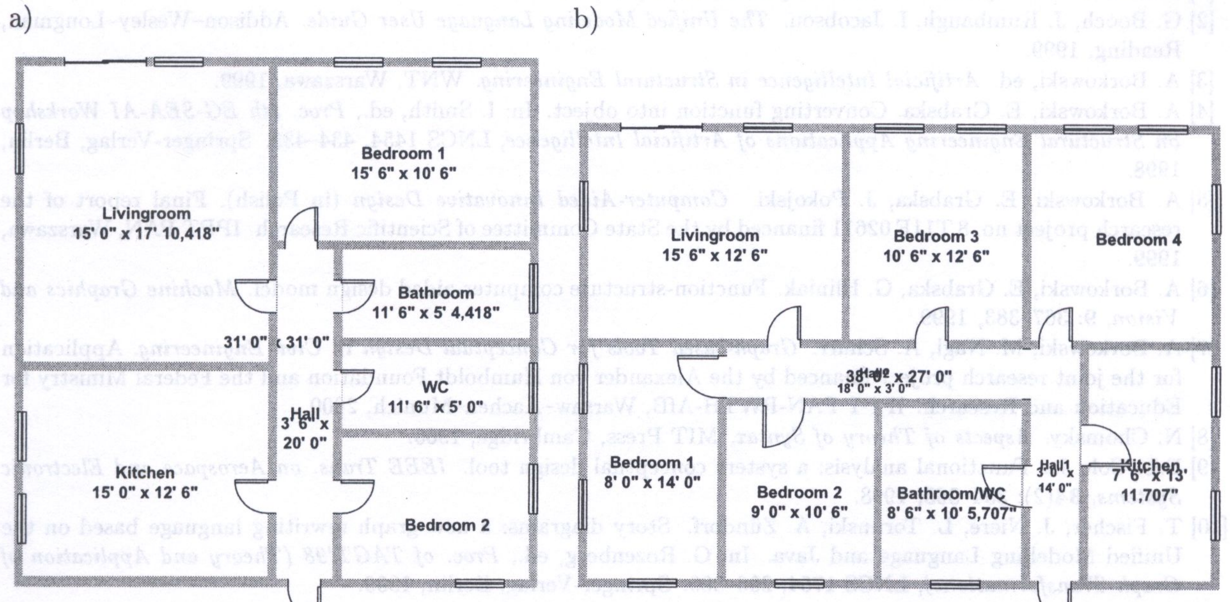


Fig. 19. Visualization of floor layout: a) obtained from structural graph in Fig. 18a; b) obtained from structural graph in Fig. 18b

Figure 19 shows two alternative floor layouts obtained by means of the prototype generator. Both solutions were derived according to the "small house" pattern. They differ in the number of bedrooms governed by the number of inhabitants and in the position of the main entrance. The latter followed from the placement of the house on the land and was fixed manually by the designer.

Usually the floor layout generated automatically serves only as a raw material for further improvement. At present all changes must be done by the architect itself within the ArchiCAD. Our aim is to equip the next version of the system with a possibility of transferring changes back to the level of graph-based functional-structural description. This will significantly improve the flexibility and assisting power of the tool.

5. Summary

In this article we have concentrated on the passage from the functional requirements of a designed object to the object structure. We restricted our consideration to a rather simple example of designing a house since the methodology remains valid for any object. In our example it is easy to distinguish various kinds of areas, rooms and relations between them and to show that graphs and graph transformations are useful as knowledge representation in computer aided design. The presented methodology seems to be appropriate for architects because they often use graphs — sometimes being unaware of it.

Acknowledgement

The research presented in the paper was partially supported by the Alexander von Humboldt Foundation and the German Federal Ministry of Education and Research (BMBF) under the joint Polish-German project "Graph-based tools for conceptual design in Civil Engineering" which is gratefully acknowledged.

References

[1] *ArchiCAD 6.5 Reference Guide.* Graphisoft, Budapest, 2000.

[2] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide.* Addison–Wesley–Longman, Reading, 1999.

[3] A. Borkowski, ed. *Artificial Intelligence in Structural Engineering.* WNT, Warszawa, 1999.

[4] A. Borkowski, E. Grabska. Converting function into object. In: I. Smith, ed., *Proc. 5th EG-SEA-AI Workshop on Structural Engineering Applications of Artificial Intelligence*, LNCS 1454, 434–439. Springer-Verlag, Berlin, 1998.

[5] A. Borkowski, E. Grabska, J. Pokojski. *Computer-Aided Innovative Design* (in Polish). Final report of the research project no. 8 T11F 02611 financed by the State Committee of Scientific Research. IPPT PAN, Warszawa, 1999.

[6] A. Borkowski, E. Grabska, G. Hliniak. Function-structure computer-aided design model. *Machine Graphics and Vision*, **9**: 367–383, 1999.

[7] A. Borkowski, M. Nagl, A. Schürr. *Graph-based Tools for Conceptual Design in Civil Engineering.* Application for the joint research project financed by the Alexander von Humboldt Foundation and the Federal Ministry for Education and Research. IPPT PAN-RWTH-AfB, Warsaw–Aachen–Munich, 2000.

[8] N. Chomsky. *Aspects of Theory of Syntax.* MIT Press, Cambridge, 1965.

[9] E.L. Cole Jr. Functional analysis: a system conceptual design tool. *IEEE Trans. on Aerospace and Electronic Systems*, **34**(2): 354–365, 1998.

[10] T. Fischer, J. Niere, L. Torunski, A. Zündorf. Story diagrams: a new graph rewriting language based on the Unified Modelling Language and Java. In: G. Rozenberg, ed., *Proc. of TAGT'98 (Theory and Application of Graph Transformations)*, LNCS 1764, 296–309. Springer-Verlag, Berlin, 1999.

[11] U. Flemming, R. Coyone, T. Gavin, M. Rychter. A generative expert system for the design of building layouts — version 2. In: B. Topping, ed., *Artificial Intelligence in Engineering Design*, 445–464. Computational Mechanics Publications, Southampton. 1999.

[12] H. Göttler, J. Günther, G. Nieskens. Use graph grammars to design CAD-systems! In: G. Rozenberg, ed., *Proc. 4th International Workshop on Graph Grammars and Their Applications to Computer Science*, LNCS 532, 396–410. Springer-Verlag, Berlin, 1991.

[13] E. Grabska. Theoretical concepts of graphical modelling. Part one: Realization of CP-graphs. *Machine Graphics and Vision*, **2**(1): 3–38, 1993. Part two: CP-graph grammars and languages. *Machine Graphics and Vision*, **2**(2): 149–178, 1993.

[14] E. Grabska. Graphs and designing. In: H.J. Schneider and H. Ehrig, eds., *Graph Transformations in Computer Science*, LNCS 776, 188–203. Springer-Verlag, Berlin, 1994.

[15] E. Grabska, A. Borkowski. Assisting creativity by composite representation. In: J.S. Gero, F. Sudweeks, eds., *Artificial Intelligence in Design'96*, 743–760. Kluwer Academic Publishers, Dordrecht, 1996.

[16] E. Grabska, A. Borkowski. Generating floor layouts by means of composite representation. In: *Proc. Worldwide ECCE Symp. on Computers in the Practice of Building and Civil Engineering*, 154–158. Association of Finnish Civil Engineers RIL, Helsinki, 1997.

[17] G. Hliniak, B. Strug. Graph grammars and evolutionary methods in graphic design, *Machine Graphics and Vision*, **9**(2): 5–13, 2000.

[18] W. Korzeniewski. *Apartment Housing — Designers Guide* (in Polish). Arkady, Warszawa, 1989.

[19] E. Neufert. *Bauentwurfslehre.* Vieweg and Sohn, Braunschweig–Wiesbaden, 1992.

[20] G. Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation.* World Science, Singapore, 1997.

[21] A. Schürr, A. Winter, A. Zündorf. Graph grammar engineering with PROGRESS. In: W. Schäfer, P. Botella, eds., *Proc. 5th European Software Engineering Conference (ESEC'95)*, LNCS 989, 219–234. Springer-Verlag, Berlin, 1995.

[22] G. Stiny. Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, **7**: 343–351, 1980.

[23] J. Szuba. *Applying generative systems in design* (in Polish). Master Thesis, Jagiellonian University, Cracow, 1999.

[24] J. Szuba, E. Grabska, A. Borkowski. Graph visualisation in ArchiCAD. In: M. Nagl, A. Schürr, M. Münch, eds., *Application of Graph Transformations with Industrial Relevance*, LNCS 1779, 241–246. Springer-Verlag, Berlin, 2000.