ISSN 1392–3730
JOURNAL OF CIVIL ENGINEERING AND MANAGEMENT
http:/www.vtu.lt/english/editions
*2003, Vol IX, No 2, 132–138*

# ON GRAPH-BASED DESIGN OF FLOOR LAYOUTS

**Adam Borkowski**[1], **Grzegorz Chaś**[2] **and Quoc Thanh Nguyen**[3]

[1] *University of Information Technology and Management, ul. Newelska 6, 01-441 Warsaw, Poland. E-mail: abork@wsisiz.edu.pl*

[2, 3] *Institute of Fundamental Technological Research, Polish Academy of Sciences, Świętokrzyska 21, 00-049 Warsaw, Poland. [2]E-mail: gchas@ippt.gov.pl; [3]E-mail: nthanh@ippt.gov.pl*

*Received 08 Jan 2003; accepted 13 May 2003*

**Abstract.** Knowledge-based tools assisting the designer in engineering represent further improvement of expert systems. The present paper shows how such software can be developed in the particular domain of floor layout design for buildings. The recently developed paradigm of hierarchical graphs is taken as the knowledge representation scheme. The user of the system is encouraged to undertake the search for rational solution at two levels. First, an analysis of functionality requirements for the designed object is performed. This results in a graph capturing main functions and relations between them. Further, this graph is mapped onto another graph depicting the floor layout in terms of areas and rooms. Both graphs produced by the user are checked against the constraints resulting from the requirements of the relevant code of practice. The final result is converted into the format accepted by a commercial CAD-tool in order to proceed with the detailed design.

**Keywords:** knowledge-based design, hierarchical graphs, graph transformations.

## 1. Introduction

Following up the popularity of MYCIN numerous expert systems were developed in the area of engineering design [1]. Unfortunately, their acceptance by practitioners was very low. The main cause of that was too stiff nature of the first generation expert systems. They forced the designer to follow a fixed path preventing him/her from coming up with innovative solutions. Thus the attempt to help designer by means of computer-based "experts" yielded similar negative effect as the attempt to standardise architecture by promoting the so-called "typical projects".

It is commonly agreed nowadays that knowledge-based design assistants who replaced expert systems should stimulate search for innovative solutions. This can be achieved by encouraging the designer to perform thorough conceptual analysis of the design task before plunging into details as well as by relieving him/her from the tedious check of the code of practice compliance of the project. The aim of the present paper is to demonstrate that both goals can be achieved within the frame of graph-oriented knowledge representation.

The theoretical framework of this representation was laid out by E. Grabska [2]. She introduced the composition graphs (CP-graphs), the realisation schemes and, later, the new model of hierarchical graphs [3]. This methodology belongs to the theory of graphs and graph transformations – a domain in Computer Science that undergoes vivid expansion in recent years (compare, eg, [4]). Graph grammars that constitute an important part of this theory can be seen as part of the linguistic approach to world modelling proposed by N. Chomsky in the 1970s [5]. 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 a generation of words and sentences define a grammar of the concerned language. In terms of engineering design 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 [6] developed the shape grammars many researchers showed how such grammars allow the architect to capture essential features of a certain style of the building (eg 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. Graphs are capable to bear much more information than linear strings or shapes. Hence, their applicability for CAD-systems was immediately appreciated [7].

In 1997–99 the formalism developed by E. Grabska was adopted for developing several intelligent design-assisting tools [8, 9]. In the present paper we report further

extension of that model. It turned out that by introducing an additional graph one can conveniently reason about the functionality of the designed object. Similar approach has been proposed in [10]. The advantage of this methodology lies in allowing 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.

Our aim is to develop prototype software that will assist an architect in the design of the layout of buildings. 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 building in terms of graphs, generates a proper graph grammar and translates the result into the input file for the commercial CAD-system ArchiCAD [12]. The architect obtains a draft layout of the building that can be visualised and presented to the investor. Further detailed design is performed in a usual way in the environment supplied by the CAD tool. The previous steps in developing the present system were reported in [13].

## 2. Graphs and graph transformations

In order to reason automatically on graphs one needs a proper tool called graph rewrite system. Several systems of that kind are available at present, mostly as non-commercial software. Due to a long-term cooperation with the RWTH Aachen, we have access to the system PROGRES developed at that university [14]. A comprehensive description of PROGRES as knowledge representation tool for conceptual design can be found in [15]. In the sequel we restrict ourselves to informal explanation of the graph-oriented methodology.

Consider a simple graph shown in Fig 1.

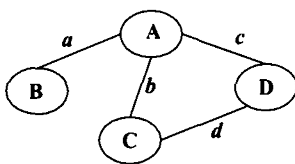


**Fig 1.** Attributed graph

The nodes of this graph carry labels and attributes. In the UML-notation [16] they correspond to the descriptions of classes. The edges are also labelled and attributed. They describe relations between particular classes of objects.

The PROGRES system can be used as a specification language. A PROGRES specification consists of two parts: the schema part and the transaction part. The first one captures static knowledge about a considered world. Here abstract classes like *PARCEL*, *BUILDING* or *ROOM* are defined together with specific node types like *SHOP* or *KITCHEN*.

Types of edges are defined for each abstract class separately. They describe relations like *contains*, *is_adjacent_to* or *is_accessible_from* that are applicable to any type of object belonging to the given class.

The transaction part captures dynamic knowledge. Within a transaction several productions and/or tests can occur. A production transforms graph $A$ into graph $B$:

$$A \rightarrow B .$$

Here A is the left hand side and $B$ is the right hand side of the production. When the production is performed, PROGRES searches in the transformed graph for all subgraphs that match $A$ and replaces them by $B$.

A test in PROGRES specification allows us to check whether certain requirements are fulfilled. Productions and tests can be combined into transactions having prescribed order of execution. Thus complex applications can be built using this graph rewrite system.

The experience gained so far with ordinary graphs manipulated by PROGRES showed that such single-level knowledge representation is insufficient in many applications. Hence, we intend to employ in the future a hierarchical graph model proposed in [3]. According to this model, the hierarchical graph is a pair $(V, E)$ where $V$ is a set of nodes $v = (i, B, C)$ and $E$ is a set of edges connecting those nodes. The $i$-th node has a set of bonds $B$ and a set of children $C$.

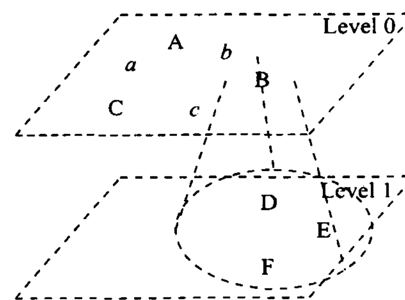


**Fig 2.** Hierarchical graph

As shown in Fig 2, nodes of the hierarchical graph represent subgraphs that can be nested to a certain depth. Such model is much more expressive than an ordinary graph. On the other hand, it requires more evolved definitions of productions and tests. At present PROGRESS does not work for hierarchical graphs. A new graph rewrite tool suitable for such graphs is under development.

## 3. Functionality analysis

Prior to designing any kind of artefact one has to know exactly what is intended for this object. In architecture this primacy of function over form was stated by the Bauhaus school in Weimar (1919–34) and later followed by many famous designers like L. Mies van der Rohe.

Functional requirements for a new building are usually determined by interviewing the investor. Questionnaires and forms used for this purpose can be found, for example, in [17]. The Unified Modelling Language pro-

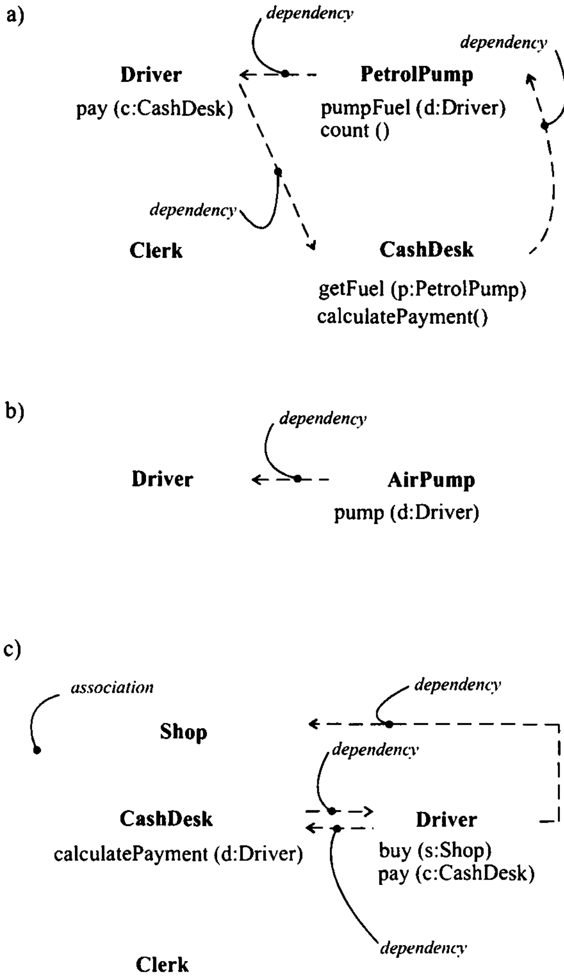


Fig 3. Use cases for petrol station: a) getting fuel; b) checking pressure in tyres; c) buying light bulb

vides use case diagrams as a convenient tool for capturing functions of the designed object.

Let us take a refuelling station as an object by which we will explain the proposed methodology. Fig 3 shows three typical situations that may occur at such a station: the user fills the tank of his car with fuel, adjusts the air pressure in the tyres or buys a spare part. The use case diagram shows the actors (the driver, the clerk), the objects (the petrol pump, the air pump, the cash desk) and the relations between them.

Based upon the use cases the designer can determine the list of functions that the considered object has to deliver. The refuelling station, taken as an example, should allow the customer to acquire fuel, to perform small maintenance of the vehicle (cleaning, checking air pressure) and to buy newspapers, food and spare parts.

The results of this analysis can be depicted in functionality graphs. The nodes of such graphs bear the names of particular functions, whereas the edges carry the labels of relations that exist between them. Fig 4 shows one functionality graph for the refuelling station. It deals with its main function – selling petrol – and includes the subordinate functions coming into play. The edges of this

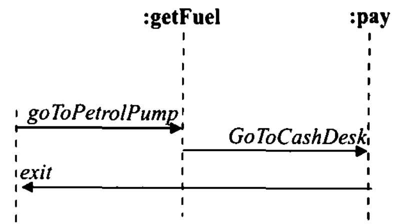


Fig 4. Function *getFuel*

graph depict the relation *followed_by* that describes the sequence of events in time.

Similar graphs can be built for the remaining functions of the refuelling station. The UML provides a rich variety of visualisation formats. For example, the synchronisation of events in time can be analysed by means of Petri nets, Gantt diagrams or Pert charts. We restrict ourselves to rather simple function graphs in order to keep the system user friendly.

## 4. Mapping functions on objects

After the functionality analysis has been completed, the designer is in the position to think about the layout of the object. Usually the process of spatial arrangement goes in a top-down manner: firstly a parcel is selected and acquired for the object, then the object is situated on the parcel and finally an internal layout of the object is determined. The management of land is currently done within the format of Geographic Information System (GIS). Graphs and graph transformations can be very useful when solving urban design problems. The development of software tools for that purpose is under way in our research group. In the present paper we restrict ourselves to the problem of finding internal layout of the object situated already on the parcel.

The top-level nodes of functionality graphs correspond to the main functions required by the object specification. It is reasonable to begin the layout design by dividing the entire area of the object into zones devoted to the main functions. In the case of refuelling station these would be the *RefuellingZone*, the *SelfServiceZone* and the *CustomerServiceZone* (Fig 5 a).

The relations between those zones are twofold: on the one hand each of them should be accessible, on the other hand the zones should be spatially separated (mainly for fire security reasons). Let us introduce an *CommunicationZone* between each pair of the main function zones. This leads to the simpler graph shown in Fig 5 b. The edges of this graph depict accessibility relations.

Having set the main zones, the user starts thinking about the components of the designed object that bear subordinate functions. In the present prototype this is accomplished by means of a manually drawn URL class diagram. In the future this part of planning will be performed by means of hierarchical graph.

**Fig 5.** Main zones of refuelling station: a) general requirements; b) spatial separation

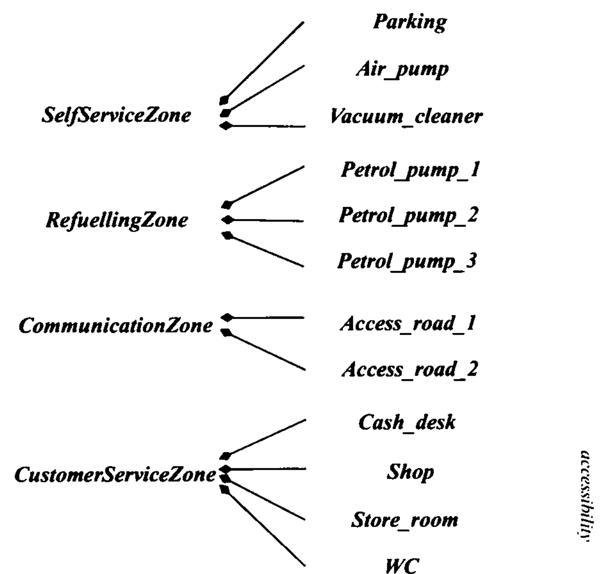


**Fig 6.** Decomposition graph: zones and compartments

A unit occupying certain region of the zone is called *Compartment*. In particular, this can be a room of the building but we treat open spaces, like access roads or parking lots, as compartments as well. Usually, a compartment carries a single function and thus belongs to a single zone. Multifunctional compartments are also possible. They lead to partial overlapping of zones on the floor layout. The present software prototype deals with single-floor objects only. The later releases will allow the design of multi-storey buildings.

Let us take our illustrative example. Fig 6 shows the compartments present at the refuelling station. The user has decided that there should be three *PetrolPumps* in the *RefuellingZone*. The *CustomerService* zone should contain *CashDesk*, *Shop*, *StoreRoom* and *WC*. The compartments belonging to the *SelfService* zone are *Compressor*, *VacuumCleaner* and *ParkingLot*. The *Communication* zone consists of a single *AccessRoad* compartment.

In our opinion, a fully automatic assignment of functions to the compartments is neither rational nor necessary. The software developed by us allows the designer to reason about the decomposition of the building into functional units prior to the detailed design phase. Graph editors supplying pictures like Fig 5 or Fig 6 enable the user to gain clear understanding of the functionality requirements and their fulfilment in the designed object. They facilitate also the dialog between the investor and the architect.

## 5. Positioning zones and compartments

After having considered the general layout of the object, the designer needs to assign the positions and the dimensions of particular compartments. This work is tedious and even a semi-automatic procedure would enhance the efficiency of the design process.

We developed a software module called *ObjectAdjuster* that fulfils this task. The main assumption is that the outer contour of the designed object is given. Additionally, each object that has to be placed inside the contour possesses the following attributes:

- *priority* – an integer number belonging to the interval *[0, 10]* that describes relative importance of this object;
- *min_area, max_area* – a pair of real numbers defining the interval of admissible values for the area of the object;
- *constraints* – a link to the rule base containing other restrictions that should be satisfied by the object.

The main idea of the *ObjectAdjuster* is to display the contour of the designed object and to ask the user to position approximately the objects that should fit inside the contour. Knowing the attributes of each object, the module tries to find the best fit and displays the result. Usually the user has to correct manually this layout. He is also encouraged to call a *Checker* module before the result is accepted.

The *Checker* has access to the graphs describing functionality of the object, as well as to the rule base containing the domain knowledge (the code of practice rules, the regional regulations, the internal rules of the design bureau, etc). Based on this knowledge, the module performs the check and informs the user whether the proposed layout conforms to all necessary requirements. If any violation is detected, a new loop of the design refinement is opened.

Let us illustrate the procedure by the example of petrol station. The first stage is the positioning of zones on the parcel. The contour of the parcel is shown by the double line in Fig 7 a. The icons of zones are visible on the screen. Dragging and dropping them inside the contour by the mouse the user selects approximate positions of zones (Fig 7 a). The dash line rectangles indicate the minimal area of each zone. In general case, these rect-

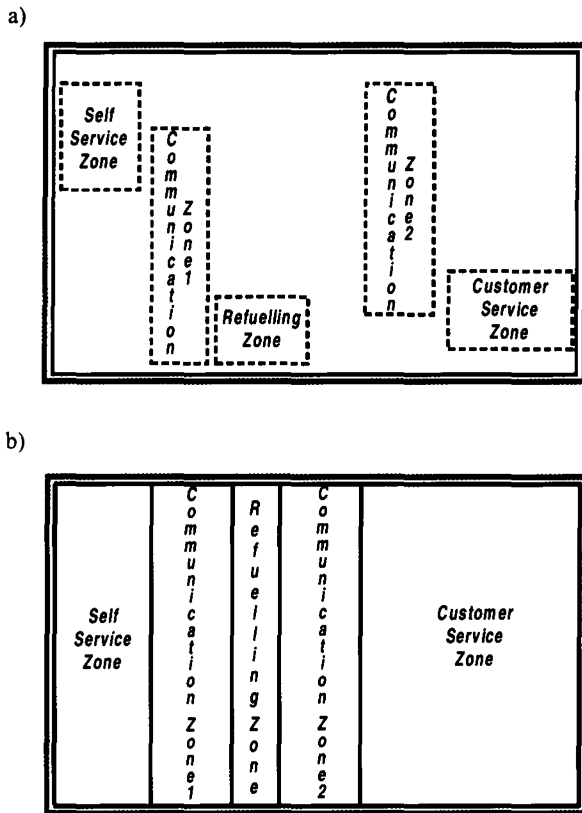


**Fig 7.** Adjusting zones: a) rough positioning; b) final positioning

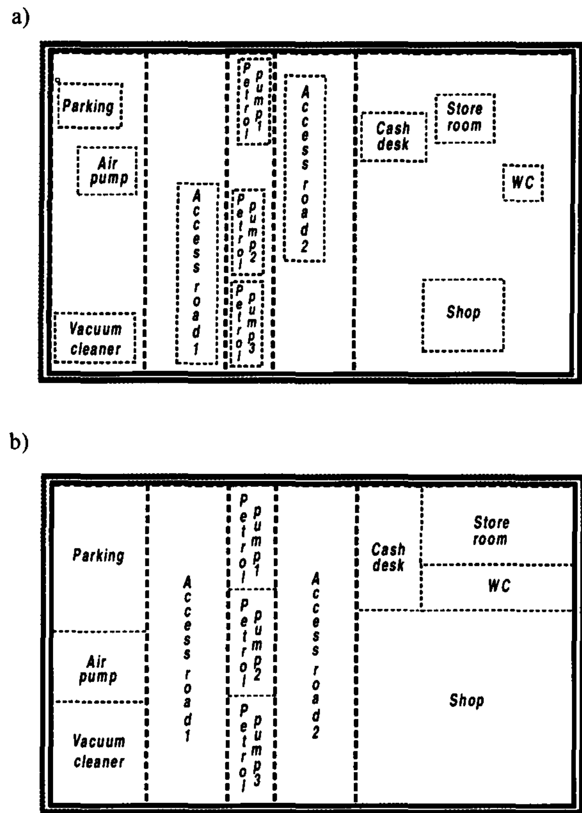


**Fig 8.** Adjusting compartments: a) rough positioning; b) intermediate positioning

angles will not necessary be disjoint as shown in the figure: they may overlap and even cross the contour.

Next the user pushes the *adjust* button and the module tries to fill the contour. If the maximal area has not been specified for a particular zone, then it is determined by the program as a part of the total area corresponding to the priority of the zone. Hence, the priorities play double role: they determine the expansion sequence of zones and, if necessary, allow the adjuster to evaluate the maximum area for each zone. The final result of zone adjustment is shown in Fig 7 b.

Now the designer may begin the second stage – adjusting compartments inside zones. It is done similarly to the first stage. The only difference is a bigger number of objects that need to be positioned. According to Fig 6, these are: the petrol pumps 1 to 3, the air compressor, the vacuum cleaner, the parking lot, the shop, the cash desk, the storeroom, the rest room and the two access roads.

Fig 8 a shows the approximate positions chosen by the user. The templates of compartments are displayed as dashed rectangles, whereas solid lines mark the contours of zones. Note that, contrary to the contour of parcel, they can be adjusted when necessary. Pushing the *adjust* button first time leads to the layout depicted in Fig 8 b. All area available was consumed but there is a fault in the accessibility of the storeroom: it should be accessible from the shop (compare relevant relation in Fig 6). The *Checker* discloses this inconsistency and the user obtains a proper error message. He must return to the previous step and move the *WC* rectangle slightly to the right.

This allows the automatic adjuster to connect the *StoreRoom* with the *Shop*. After minor adjustments by hand the final layout shown in Fig 9 is obtained.

The final product of the *ObjectAdjuster* is the sketch of layout. It is accompanied by the data like the areas of compartments, their priorities and functions. In order to perform a detailed design the architect needs to transfer this information into a commercial CAD tool. The next section deals with this problem.

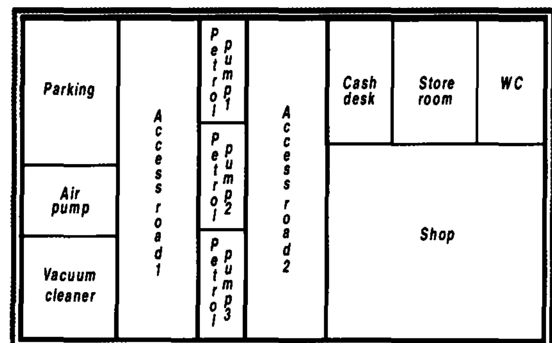


**Fig 9.** Adjusting compartments: final result

## 6. Linking to ArchiCAD and visualising

In this section we describe a XML-based mechanism for translating results obtained by the GraCAD into the ArchiCAD – one of several professional CAD-systems available at present. XML is a good choice as it is a standard data representation format and it can be easily adopted for different environments. XML documents are produced for communication between ArchiCAD and other applications in two ways. For graph visualisation purposes, the XML document contains all information about a graph. For graph updating, another XML file contains information about changes that have been made to elements in ArchiCAD or transformations that should be made on a graph.

Graphs can be easily represented by XML. It suffices to define a collection of tags representing nodes, edges and transformations of a graph, as well as basic construction elements such as *WALL*, *DOOR* and *WINDOW* that are present in the CAD-system. We need them in order to visualise adjacency and accessibility relations. A tag for node or construction element contains a list of attributes-value pairs. Typically, such list includes the identifier, the label, the type and several geometrical attributes. A tag for edge has a list of all nodes linked by that edge. A tag for graph transformation includes a list of subgraphs which the transformation should apply to. A part of the XML-file describing functionality graph is given in Fig 10.

```
<Zones>
      <Zone id='RefuellingZone1'
label='RefuellingZone1'
type='0'/><Zone id='SelfServiceZone1'
label='SelfServiceZone1'
type='1'/
<Zone id='CustomerServiceZone1'
label='CustumerServiceZone1'
type='2'/>
      <Zone id='CommunicationZone1'
label='CommunicationZone1' type='3'/>
      <Zone id='CommunicationZone2'
label='CommunicationZone2' type='3'/>
</Zones>

<Relations>
      <ZoneAccess
zone1='RefuellingZone1'
zone2='CommunicationZone1' />
      <ZoneAccess
zone1='SelfServiceZone1'
label='CommunicationZone1' />
      <ZoneAccess
zone1='RefuellingZone1'
zone2='CommunicationZone2' />
      <ZoneAccess
zone1='CustumerServiceZone1'
zone2='CommunicationZone2' />
</Relations>
```

**Fig 10.** XML tags of the functionality graph

In the ArchiCad environment the user can either visualise the decomposition graph or the layout itself. In the first case, the user can modify the graph by adding new node, deleting existing node, changing geometrical attributes of a node and then start the adjustment process. The XML document describing the graph is updated automatically after each change of the graph. In the second case, the XML document contains the description of the final project.

In the ArchiCAD environment, XML documents are interpreted by a DLL-module – an add-on implemented in C and linked to the system. This add-on defines a class of objects called *COMPARTMENT* that is able to visualise a node of the decomposition graph. The object *COMPARTMENT* is linked to the basic elements of the ArchiCAD library of primitives: walls, windows, doors, etc. *COMPARTMENT* objects can be created, moved, changed or deleted. After each operation of this kind the add-on generates

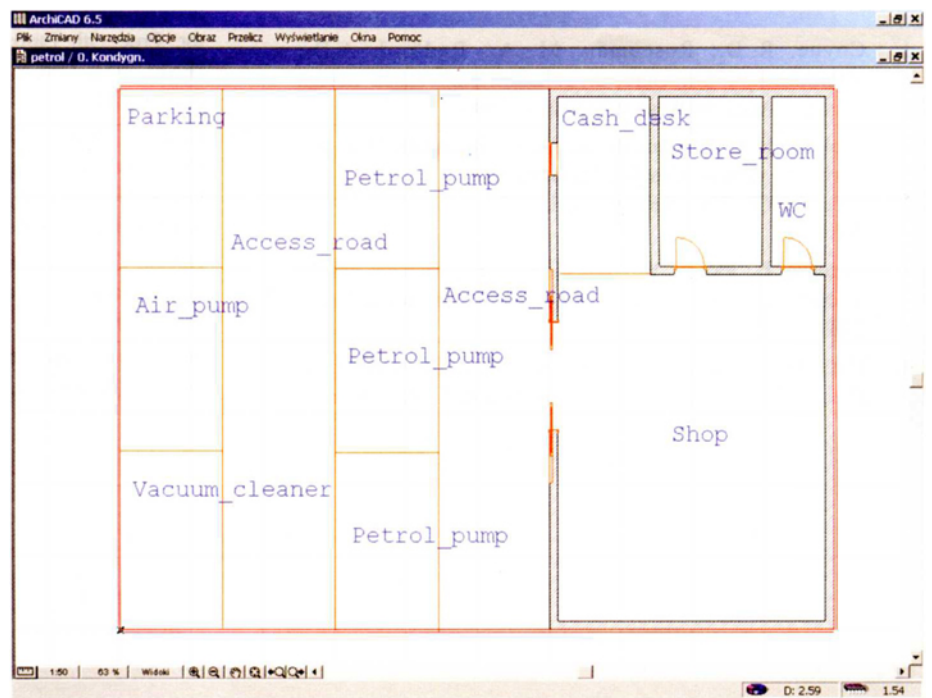


**Fig 11.** Petrol station – final solution

the XML document, translates it into an internal format of the ArchiCAD and allows the user to visualise the project. Fig 11 presents the final solution for petrol station.

The main obstacle that has to be overcome when connecting graph-oriented software to commercial CAD-systems is the discrepancy in the granularity of knowledge representation. Most CAD-systems work with primitives that are far lower in abstraction than functionality graphs, decomposition graphs or UML-diagrams. In particular, the ArchiCAD uses *WALL* as the basic primitive. Placing two adjacent rooms on the layout according to the graph-oriented description would result in duplicate walls in the ArchiCAD. Therefore, a special method was implemented in the add-on. It merges a pair of duplicate walls into a single one and, moreover, allows us to introduce virtual walls, like those separating *CommunicationZone* from *SelfServiceZone*, *RefuellingZone* and *CustomerServiceZone*.

## 7. Conclusions

Graph-based knowledge model is expressive enough for the floor layout problem in Civil Engineering. Its applicability in other areas, like Machine Building or Electrical Engineering, seems to be worth consideration. Graph rewrite tools available today allow us to develop flexible design assistants. Thus it seems that the barrier of "frozen knowledge" precluding wider applicability of conventional expert systems has been overcome. Further research is needed in the area of hierarchical graphs in order to exploit fully their potential capabilities.

## References

1. Coyne, R. D.; Rosenman, M. A.; Radford, A.D.; Balachandran, M.; Gero, J. S. Knowledge-based design systems, Addison-Wesley, Reading, 1990.
2. Grabska, E. Graphs and designing. In: Schneider H. J. and Ehrig H., editors. Graph transformations in computer science, LNCS 776, Springer-Verlag, Berlin, 1994, p. 188–203.
3. Grabska, E.; Palacz, W.; Szyngiera, P. Hierarchical graphs in creative design, *Machine GRAPHICS & VISION*, Vol 9, 2000, p. 115–122.
4. Corradini, A.; Ehrig, H.; Kreowski, H.-J.; Rozenberg, G. (Eds.). Proceedings of the 1st international conference on graph transformations (ICGT'02), October 2002, Barcelona, Spain.
5. Chomsky, N. Aspects of theory of syntax, MIT Press, Cambridge, 1965.
6. Stiny, G. Introduction to shape and shape grammars, Environment and planning B. *Planning and design*, 7, 1980, p. 343–351.
7. Göttler, H.; Günther, J.; Nieskens, G. Use graph grammars to design CAD-systems! In: Rozenberg G., editor, Proc. 4th International workshop on graph grammars and their applications to computer science, LNCS 532, Springer-Verlag, Berlin, 1991, p. 396–410.
8. Borkowski, A.; Grabska, E. Converting function into object. In: Proceedings of the 5th EG-SEA-AI workshop on AI in structural engineering, ed by I. Smith, Ascona, July 1997.
9. Borkowski, A.; Grabska, E.; Hliniak, G. Function-structure computer-aided design model. *Machine GRAPHICS & VISION*, 8 (1999), p. 367–383.
10. Cole, E. L. Jr. Functional analysis: a system conceptual design tool. *IEEE Trans. on aerospace & electronic systems*, 34 (2), 1998, p. 354–365.
11. Flemming, U.; Coyone, R.; Gavin, T.; Rychter, M. A generative expert system for the design of building layouts – version 2. In: B. Topping, ed., Artificial intelligence in engineering design, computational mechanics publications, Southampton. 1999, p. 445–464.
12. ArchiCAD 6.5 Reference guide, Graphisoft, Budapest, 2000.
13. Szuba, J.; Grabska, E.; Borkowski, A. Graph visualisation in ArchiCAD. In: Nagl M., Schürr A., Münch M., editors. Application of graph transformations with industrial relevance, LNCS 1779, Springer-Verlag, Berlin, 2000, p. 241–246.
14. Schürr, A.; Winter, A.; Zündorf, A. Graph grammar engineering with PROGRESS. In: Schäfer W., Botella P., editor. Proc. 5th European software engineering conference (ESEC'95), LNCS 989, Springer-Verlag, Berlin, 1995, p. 219–234.
15. Kraft, B.; Meyer, O.; Nagl, M. Graph technology support for conceptual design in civil engineering. In: Schnellenbach-Heldt M. (Ed.). Proc. of 9th Int. Workshop on intelligent computing in engineering, Darmstadt, August 2002.
16. Booch, G.; Rumbaugh, J.; Jacobson, I. The unified modeling language user guide. Addison Wesley Longman, Reading, 1999.
17. Neufert, E. Bauentwurfslehre. Vieweg & Sohn, Braunschweig-Wiesbaden, 1992.