EG-ICE-2002 - Int. Workshop of the European Group for Intelligent Computing in Engineering
eds. Schellenbach-Held, M., Denk, H., VDI Fortschritt Berichte, (180), Seite 1-35, VDI-Verlag, Darmstadt

GRAPH TECHNOLOGY SUPPORT FOR CONCEPTUAL DESIGN IN CIVIL ENGINEERING

Bodo Kraft, Oliver Meyer, Manfred Nagl

Department of Computer Science III, RWTH Aachen
D-52074 Aachen

{kraft, omeyer, nagl}@i3.informatik.rwth-aachen.de

Abstract: The paper describes a novel way to support conceptual design in civil engineering. The designer uses semantical tools guaranteeing certain internal structures of the design result but also the fulfillment of various constraints.

Two different approaches and corresponding tools are discussed: (a) Visually specified tools with automatic code generation to determine a design structure as well as fixing various constraints a design has to obey. These tools are also valuable for design knowledge specialist. (b) Extensions of existing CAD tools to provide semantical knowledge to be used by an architect. It is sketched how these different tools can be combined in the future.

The main part of the paper discusses the concepts and realization of two prototypes following the two above approaches. The paper especially discusses that specific graphs and the specification of their structure are useful for both tool realization projects.

1. INTRODUCTION

In this chapter we first describe some projects ongoing in our group, before we introduce our approach for supporting conceptual design. After that we discuss related work and give an overview of the paper.

1.1. Some ongoing projects

Department of Computer Science III at RWTH Aachen is specialized in software engineering and software architectures. Especially, we have a broad experience in building new tools for various application domains[24; 25]. In the following some *ongoing projects* are sketched, because of two reasons. On one hand, the projects describe the realm of our activities in different domains. On the other, as we shall see later, these projects share some similarities – either in their underlying concepts or their realization – with the project introduced in this paper.

Within the *AHEAD project* an integrated management system for development processes is investigated [17; 31; 36]. In this context, management denotes the coordination of cooperating developers, the administration of their complex product, as well as the assignment of resources to tasks. The management system works on administration level, and, therefore, above the level how technical developers perform their tasks or how their results look like. The management system tightly integrates submodels dealing with processes, products, and resources, respectively by offering corresponding tools. Furthermore, the system can be parameterized for a specific context (application domain, standard, company etc.).

Tool support on technical level is given by *integration tools*, having been investigated for software development some time ago [20; 24] and being currently investigated for chemical engineering [3]. These tools help to keep the contents of different documents consistent to each other. These interactive and incremental tools establish and maintain inter-document links by mutually relating increments of different documents. Changes of the underlying documents are recognized, the integration tools help to transform these changes, to analyze different documents for consistency, and alike. Integration tools avoid the disadvantages of batch-oriented transformators and of manual link-editing tools.

The *CHASID project* aims at supporting authors of textbooks, scientific texts, and articles by modeling the semantical structure of text documents [11]. A topic map describes and relates topics the author whishes to communicate. It is integrated with the hierarchical structure of the text.

Templates on the level of the topic map, or the structure of the text, or their interrelation yield an improved structure of the text. In the same problem field the *aTool project* extents MS Word by semantical features [23].

The *E-CARES project* develops concepts, methods, and tools to support the processes of understanding and restructuring complex legacy telecommunication systems [15]. To provide a better understanding of existing systems, their current statical structure, their monitoring data at runtime, as well as “method rules” of its developers are regarded. The project unites a top-down approach developing a system description language and formulating the future structure of a system in that language, and a bottom-up approach determining the existing structure and corresponding reengineering tools. The used tool construction concepts are similar to another project dealing with reverse and reengineering of business administration applications [8].

1.2. Conceptual design as it is and as it should be

Today most of the architects do not elaborate their sketches inside a CAD system, they rather work with pencil and paper. In the early design phase, the architect does not think about materials or realization details, he works creatively by sketching room dimensions and positions. The design of the future building is still vague and imprecise. *Constructive elements*, like walls, windows, or doors are used with their *conceptual meaning*, namely to form rooms, to guarantee light and ventilation, or to ensure accessibility. These conceptual elements, therefore, form a functional view of the architectural design structure which , however, is *not explicitly* determined.

When the architect has finished his sketch he manually creates a constructive design within a CAD system. Now he *replaces* the *functional* elements of the sketch by *constructive* parts, as the ventilation by a window, the access by a door etc. The conceptual information he has in mind gets lost by the time.

There are many *changes* within the development process. For example, if a cost calculation is elaborated after the detailed design, this might imply quite another structure of the building. In the same way, if the client is not

satisfied with the design, the architect has to go back to restart the conceptual design process. After the redesign of the conceptual sketch, the changes have to be transferred again manually into the CAD system. The *modification* of the *conceptual data* are *lost* again. These iterations, which might be repeated some times, cost time and money, but also increase the risk of errors.

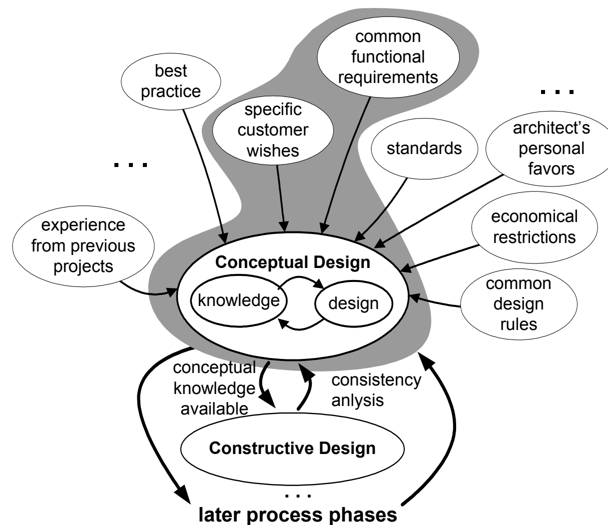


Fig. 1: Design constraints for conceptual design and later phases

In principle, the architect is *constrained* by different dimensions in his *design* process. There are standards to fulfil, the requirements of the customer have to be met, the personal design favour of the architect has to be obeyed, to name only three of these dimensions. Fig. 1 depicts this situation. [29; 30] argue that various and different aspects have to be taken into consideration for the whole development process.

Our approach is to avoid the expensive backtracking steps in the development process by *explicitly* determining the structures and restrictions of conceptual design and by *using* this knowledge in conceptual design and later phases. The reason is that errors are cheaper if detected as early as possible.

From the list of dimensions of Fig. 1 we *focus* on common functional requirements and specific customer wishes in this paper (grey shaded area). The presented argumentation holds true for the other dimensions as well.

As is indicated in Fig. 1, we explicitly determine the result of conceptual design. We also explicitly determine the underlying concepts and constraints due to the above dimensions. Then, we *check* the current sketch (conceptual design result) against that explicit knowledge. Furthermore, this conceptual knowledge is also used in *later phases* of the design process. In this paper we only address the aspect of using design knowledge in constructive design, following the conceptual design.

Summing up, to improve the design process of the architect, five topics are essential:

1. The *conceptual* design process should be supported by *tools*, explicitly determining the resulting conceptual design.
2. The underlying *structure*, as well as the *constraining knowledge* have to be *explicitly* formulated.
3. For determining the underlying *knowledge*, corresponding *tools* also have to be developed.
4. The conceptual design result should easily be *transformed* for *later* process phases. For example, it should be possible to generate a preliminary floor plan to show it to the customer.
5. Also, the conceptual information should be used to make *various consistency checks* for later process results, especially for the developed floor plan.

The *running example* of this paper is the design of a single-family house. Of course, there are specific subtypes of this category. Our explanation will not go deep enough to distinguish these subtypes.

1.3. Related work

There exist different approaches to support the architect during design. Most of them use concepts from the field of artificial intelligence. We roughly separate them in *evaluative* and *generative* approaches [21].

One form of evaluative approaches use a *knowledge base* to store information about a specific domain and design decisions made in the past together with their context. The system filters those parts from the knowledge base that are important for a current problem of the architect. The systems support human decision and, therefore, are called *Decision Support Systems* (DSS). To know which part of the knowledge base is relevant, the design in most cases strictly follows a predefined process model (*Case Based Design*) [1; 22; 27; 29; 30; 33].

Another form of evaluative approaches do a *technical analysis* of the constructive design. Whereas simulations like daylight analysis, stability analysis etc. just use numerical calculations, other aspects – like the fulfillment of customer needs – use again knowledge bases [6; 7].

Generative approaches mostly use the represented structural knowledge to *create* at least an initial *prototypical design*. Parts of that design are then further refined, by applying generative rules [10; 35; 37]. In these approaches it is mainly the machine that creates the design. *Shape grammars* [12; 13] also make use of rules for detailing steps in design. These rules, however, are interactively applied during design.

Most of the above mentioned approaches are implemented in Prolog or Lisp and are not integrated with existing CAD systems. [14] describes another graph-based generative approach to support design. A *graph rewriting system* for planning the layout of a kitchen from a graph structure is presented. The conceptual ideas are similar to those presented in the paper. However, the approach for structuring conceptual knowledge is simpler and there is no connection to existing systems.

Current *CAD systems*, like AutoCAD architectural desktop, have *extensions* to introduce concepts like rooms into a given design. However, these concepts are not on conceptual level, they are only used to a-posteriori mark areas as rooms. Some calculations can use these marks. The conceptual feature cannot be used during the design process by the architect.

One year ago, we have started a new joint project (with A. Borkowski, Warsaw, E. Grabska, Cracow, A. Schürr, Munich), the aim of which is to *support* architects in *conceptual design* of houses. [4; 34] model the

functionality of specific areas of a house by a graph structure. From that structure a graph of related rooms with their positions is generated automatically as a suggestion for the architect which forms a floor plan. In this paper we rather follow an interactive approach where the designer is dominating the design process.

1.4. Overview of the following paper

In this paper concepts and tools answering the five problems of 1.2 are presented. As graphs, their formal description and corresponding realization techniques play a big role in this paper, we first give an overview of our experience, called graph technology in section 2. Section 3 describes our approach to explicitly deal with conceptual knowledge on one hand and to use this knowledge for conceptual or constructive design on the other. Two prototypes are discussed, together with their future integration to form an overall solution. Section 4 and 5 describe these two prototypes. Section 4 deals with the underlying concepts and the realization of a tool for explicitly structuring conceptual knowledge and the corresponding constraints and to use this knowledge during conceptual design. Section 5 discusses an extension of the commercial system ArchiCAD which introduces semantical features for that tool. The paper ends with future plans.

Summing up, our approach belongs to the group of evaluative systems. It does not enforce the user to follow a specific design process. The result of a conceptual or constructive design is constantly checked against the predefined structures and constraints.

2. GRAPH TECHNOLOGY

Common to most of the projects of our group is the use of *graph technology.* We formally model an application domain *to be investigated* by a graph class. Each graph of that graph class then can be understood as a suitable structure of the application domain. By analyzing the graph we can derive inconsistencies or problem spots and point the user to them. User interactions, for example, to build up graphs or to correct inconsistencies,

are expressed as transformations on the graph. Analyses and transformations are explicitly specified.

In this section we give a sketch of this technology based on a simple example of conceptual design. The example as well as the proceeding is simplified, as we shall see later. Graph technology uses the graph specification language PROGRES [32] and a realization machinery to easily implement tools following a specification in that language.

2.1. PROGRES as specification language

The underlying notion for describing a graph class are directed, typed, and attributed graphs. A *PROGRES specification* consists of two parts. The schema part describes the node and edge types of the graph class under consideration. The transaction part formulates transformations and tests, which can be build up to form complex transactions. We now explain both parts using our simple single-family house example. Transactions and their constituents have to be consistent with the schema part.

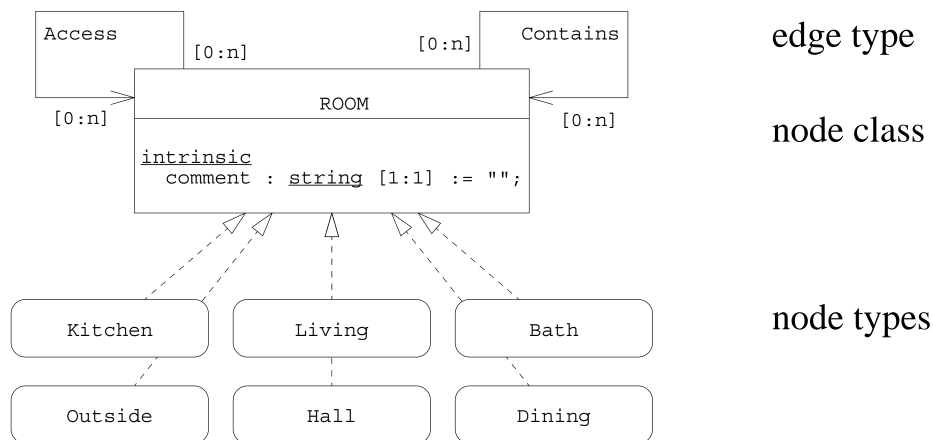


Fig. 2: Schema of our single-family house example

The *schema part* of our example is shown in Fig. 2. It shows the abstract node class `ROOM` with a `comment` attribute. Nodes of that class can be related through *`Access`*- and *`Contains`*-edges. The node class is specialized into

six different node types that represent the different kinds of rooms we model. Therefore, the relations can connect rooms of all specific types. We see that a node class expresses the similarities of different node types.

The *transaction part* determines how different graphs of that graph class are built. Within transactions productions and tests can occur. A production defines a change operation for a graph. It consists of a left and a right hand side. The left hand side is a graph pattern that describes a subgraph and the right hand side is a graph which is embedded at the place where the graph pattern's match is removed.

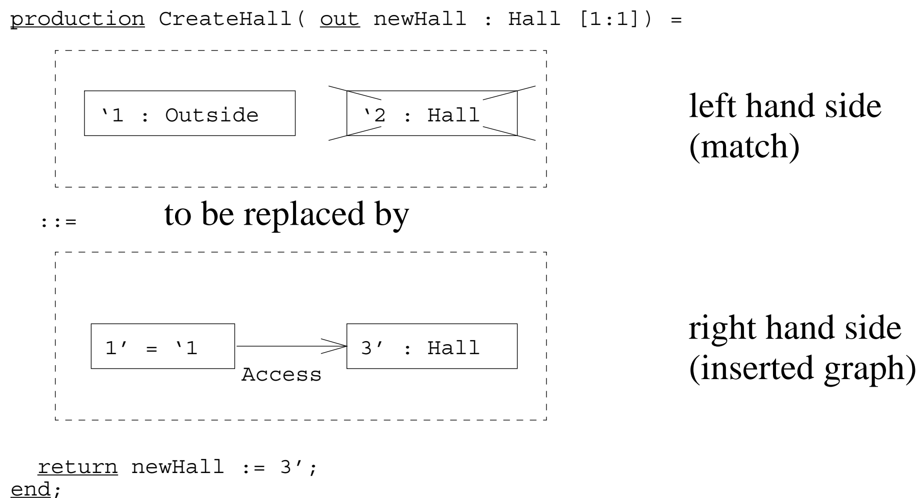


Fig. 3: Example of a graph production, inserting a node and an edge

Fig. 3 shows a *sample production* for our example. The graph pattern to be searched requires an `outside`-node to exist and no `hall`-node to be already present (negative application condition). If this pattern is found in a graph, when applying the production, a new `hall`-node is created and connected with the outside by an *`Access`*-edge. So, the application of the production guarantees that the hall is always directly accessible from outside.

Thus, each graph of our example specification models the structure of a floor plan. A `ROOM`-node without an *`Access`*-relation stands for an inaccessible room. The PROGRES *test* shown in Fig. 4 finds such rooms. It

searches the graph for `ROOM`-nodes which are not connected with the outside by a sequence of *`Access`*- or *`Contains`*-relations. The result is a possibly empty node set. In this way we formally define the meaning of “inaccessible”.

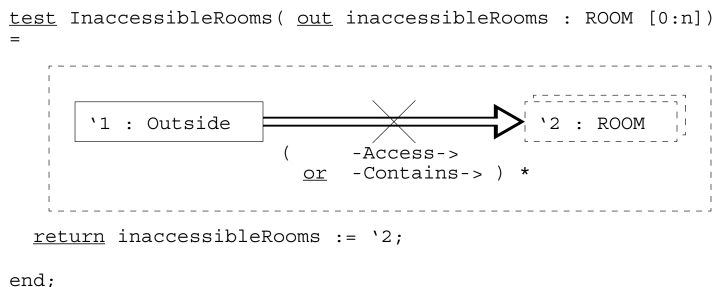


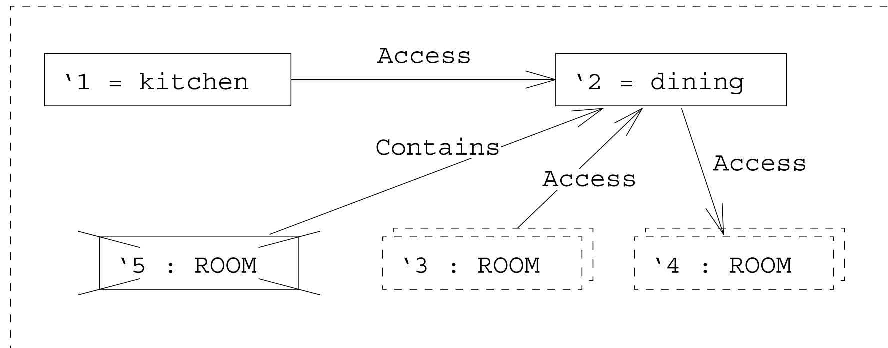
Fig. 4: Example of a graph test, finding inaccessible rooms

Graph productions may be regarded as simple user actions to edit a graph, thereby modeling a floor plan. There can also be productions that remove or replace nodes or edges from the graph. Fig. 5 shows a production to *transform* a separate *dining room* into a separate part within the kitchen. It takes these separate rooms as parameters, here `kitchen` and `dining`. The parameters can be chosen by user selection. For the production to be applicable, the nodes must be connected by an *`Access`*-edge. Also, the dining room must not already be contained in another room. As the role of the `dining`-node changes from a separate room to a separate part of another room, *`Access`*-edges that are connected to it are redirected to the `kitchen`-node.

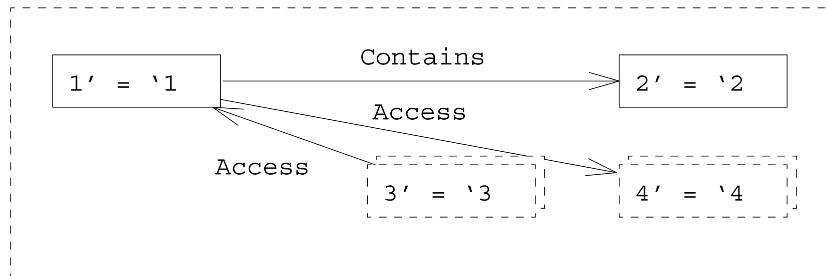
More complex user actions can be defined by applying multiple productions in a defined order. PROGRES offers control structures to combine production applications and tests to form *transactions.* They can be regarded as complex user interactions. Transactions may use applications of other transactions, too.

A *graph rewriting system* consists of a graph schema and a set of transactions. Transactions combine productions, test and inner transactions. This combination is described by control structures. The graph rewriting system is described by a PROGRES specification describing a graph class.

```
production DiningInKitchen ( kitchen : Kitchen [1:1] ;
                             dining : Dining [1:1])
=
```




```
  ::=
```




```
  folding { ‘3, ‘4 };
end;
```

Fig. 5: A graph production rearranging dining

The graphs of the graph class are the result of applying transactions. Transactions and there parts are consistent with the schema.

2.2. Implementing tools according to a specification

The *PROGRES system* [32] is an environment for developing PROGRES specifications. It is a visual programming environment as it allows to describe the schema as well as the transaction part graphically, exactly as shown in the figures above. By the editor tool, specifications can be built up, the analysis tool is checking the consistency of the specification, especially with the schema, an interpreter is used to execute a specification and to debug it. The different tools of the environment are tightly integrated.

So, during debugging, you might re-edit, immediately check, and restart the execution.

From a tested specification an *interactive tool* can be *automatically generated:* From the PROGRES specification, C-code is generated. This code makes use of a given library (PGC) that implements PROGRES' dynamic semantics. Together with the UPGRADE framework [5; 16] a tool with a graphical user interface can be easily realized. The framework consists of all reusable parts of tools. The generated C-code is plugged into that framework. This is described in more detail in this subsection. The graph prototype described later in section 4 is realized in that way.

Fig. 6 describes the structure of such a prototype. The *UPGRADE framework* is written in Java and uses the C-code generated by the PROGRES system as its main functional component. From the UI-part a user can interactively execute commands for all defined transactions and productions, thereby editing a model.

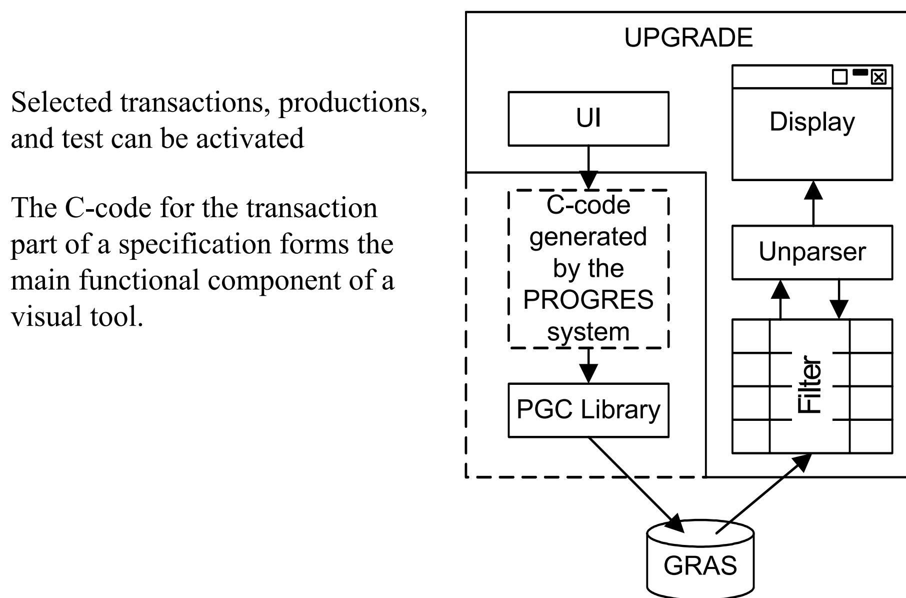


Fig. 6: Building tools by inserting generated C-code into the UPGRADE framework

The executed code due to transactions uses the *PGC library* that stores all graphs it transforms in the non-standard database *GRAS* (GRAph Storage [18]). Each change of the database sends events to the UPGRADE framework. A *filter stack* transforms the events into a form which is “understandable” in the application domain. The *unparser* receives the filtered events and updates the display accordingly. To create a graphical layout on the display it queries the graph database by means of the filter stack to retrieve the context of a new node or its attribute values. On the display the user can select *nodes* to use them as *parameters* for other transactions he wishes to execute.

We now discuss the *command cycle* by an example. For the `DiningInKitchen`-production, the user selects a `kitchen`- and a `dining`-node and then selects the transaction to be executed. The UI determines parameters for the generated code from the selection and calls the corresponding function of the generated code. Then, the corresponding transformations on the database are executed which, in turn, inform the filter stack about new and redirected edges. The event passes the stack unchanged, as *`Access`*- and *`Contains`*-edges are displayed. The unparser queries the filter stack for source and target node of the new edge, calculates their corresponding graphical object and informs the display about a new line (as a representation of that edge) betweens these objects.

The UPGRADE framework can be *parameterized* to adapt the prototype to specific needs. To define the UI behavior, the tool creator can restrict the set of transactions offered to the user, define icons for them, and arrange them in command bars and menus. To present an easier understandable model to the user, the tool creator can also parameterize the filter stack. Especially, he defines a cutout of the complete graph, based on node types or attribute values and he re-presents simple graph patterns into attributed edges not directly provided by PROGRES. Furthermore, parameters of the display can be used to define the shape, color, and the annotations of nodes and edges, based again on type or attribute values. All these parameters are stored in a single XML file defining the appearance of the tool.

The tool creator can, moreover, also *extend* the UPGRADE framework *by programming* new interaction elements, filter types, specific unparsers, representation classes or display types in Java. This allows arbitrary flexibility when building a prototype with, however, considerable higher costs. The tool described later does not make use of these extensibility features.

2.3. Extending or integrating existing tools

Both subsections of above describe a *top-down approach*. From a PROGRES specification a visual tool is realized with minor effort using the UPGRADE framework. This tool is stand-alone prototype, to be used for proof of concept purposes. As explained, it needs a rather complex infrastructure (GRAS, code generator, PGC code library, and the reusable components of UPGRADE). It can be used in academia but hardly for industrial purposes.

In industrial practice, many tool exists, especially in engineering disciplines. They have a tremendous economic value due to their development costs, the experience of their users, their usage in industrial projects and, finally, the value of systems developed by them. Therefore, about half of the projects of our group follow a *bottom-up approach* to extend existing tools or to integrate such tools. In any case, it is our aim to add new user functionality.

We give *some example*: The integrator project described in 1.1 integrates a flow chart and a simulation tool. The aTool project extends MS Word by new semantical functionality. The same is true for CHASID which extents ToolBook.

By extending or integrating tools two different tasks have to be solved: On one hand, *wrappers* are written in order to uniformly access the functionality of a tool and its underlying data. This is a prerequisite for integrating tools in a system technical sense such that these tools can be activated in a homogenous way. Building new functionality on top of tools or integrating tools in the sense of giving additional functions for developers means to create *new* data structures and new tool *functionalities*.

These data structure and tool behavior are again regarded to consist of graphs and graph transformations. So, again *graph technology* can be applied.

Extending tools can be realized in two ways: One way is to *couple* an existing tool with a PROGRES/UPGRADE prototype. Here, the coupling has to be realized, the remaining task is as described above. Trivially, the disadvantage of above also applies here, namely that the complex infrastructure has to be used. The second way is to *extend* an existing *tool* by using the extension mechanisms of the tool itself. If the new functionality is specified by PROGRES, then this functionality firstly can be approved by a visual prototype and later be transformed into the extending program. In this case, specifications are hand-coded as it was in the early days of applying graph rewriting specifications in our group (see the programming in the small tools in [24]).

3. NEW TOOLS FOR ARCHTIECTURAL DESIGN

Today, only the constructive design process is supported through various CAD systems. Defining *tools* for *conceptual design* and having corresponding explicit knowledge at hand when elaborating the constructive design, especially allows to constantly *check* the *consistency* of a constructive design against this knowledge. If, furthermore, a tool for constructive design is integrated with the conceptual design tool, we can also check the consistency with conceptual design knowledge at constructive design level. This allows rapid prototyping, reduces the number of design errors, and increases the quality and efficiency of the design process and product.

The following approach to support design in civil engineering consists of two different parts. In the first part (*top-down*) the possibilities of direct graph technology support in the sense of a *graph-based* PROGRES/UPGRADE *prototype* are investigated. This is used to offer tools to explicitly define the underlying structures and constraints of conceptual design. Furthermore, we also define the tool operations to be

used in conceptual design. These operations check for the consistency of the current conceptual design structure with the conceptual design knowledge explicitly defined before. For this part the existing PROGRES/UPGRADE infrastructure is used as described in the previous section. However, the proceeding is more complicated than described in the graph technology section.

The second approach uses the experiences of the *bottom-up* projects, namely to *extend* existing tools by adding further semantic functionality. In our case the commercial CAD system *ArchiCAD* is taken. The idea is that the architect develops buildings with ArchiCAD in a conceptual way. To avoid design errors his design results are supervised by a consistency checker using design knowledge encoded within the ArchiCAD extension. This knowledge deals with technical and law restriction, but also economical and personal decisions [2; 9; 19; 26; 28].

In the third subsection we sketch that both parts of above can be integrated to form an *overall solution*.

3.1. Tools for denoting and using conceptual knowledge

This subsection describes the first part of our approach. The realization is based on top-down graph technology as explained in subsections 2.1 and 2.2. Fig. 7 shows the structure of our *specifications* and the involved *graph structures* in a schematic view. Rectangles represent graphs, while circles represent parts of an overall PROGRES specification, each specification part being separated into a schema and corresponding transactions.

The graph in the lower right corner is the so called *room instance graph*. It is a graph of the class of conceptual design graphs as described in the last section.

The *room type graph* on the upper right corner represents the conceptual design knowledge as indicated in Fig. 1. This design knowledge has to reflect the various dimensions of a design process. The different parts of knowledge according to these various dimensions are united in this room type graph.

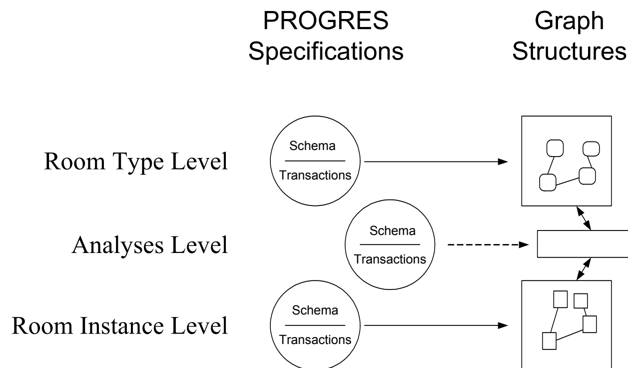


Fig. 7: Graphs and specifications of the top-down prototype

The room instance graph is *incrementally checked* against the room type graph in order to guarantee the consistency of the conceptual design with the corresponding knowledge.

For the room type graph, the room instance graph, and the analyses between both corresponding specifications exist (see left hand side of Fig. 7). The *specification* of the *room type graph* reflects the expressiveness of describing the structure and constraints of conceptual design. For example different room types can be expressed as, e. g., `hall` and `outside` and, furthermore, that both have to be connected.

The *specification* of the *room instance graph* defines the available commands for conceptual design. For example, rooms of a certain type can be inserted and interrelated.

The *analyses specification* specifies the analysis operation between room instance graph and its corresponding room type graph.

A room type *graph* is a *member* of the room type graph *class defined* by the room type *specification*. Accordingly, a room instance graph belongs to the room instance graph class given by the room instance graph specification. Finally, the analyses between room instance graph and room type graph are in accordance with the analyses specification.

The reader might have recognized that the specification explained here looks differently than the specification presented in the last section. As this is no workshop in the graph rewrite community we only present a coarse

explanation. We see that a *specification* is given here on *two levels*. Whereas in the PROGRES specification given in the last section the conceptual knowledge is hard-coded in the specification, the new proceeding offers a level and corresponding tools by which this knowledge can be explicitly put in. The reason is, that in design in civil engineering we can hardly find PROGRES specialist. Even more important, however, is that this knowledge can be changed and extended.

According to this two-level approach the specification of the room instance graph only consists of *generic operations* like inserting/deleting rooms or connecting them by edges. The complex *restrictions* underlying such operations are not to be found within a specification (as in the last section) but in the *room type graph*. Accordingly, the consistency constraints to be found in the PROGRES specification are now in the explicit description of the room type graph and the analyses checking whether a room instance graph is consistent with a room type graph.

The two-level specification approach now yields an UPGRADE *prototype* with *two views*. By the room type view we find commands for creating conceptual knowledge. These commands are as specified in the room type specification. The room instance view offers command for building up a conceptual design. The second view is either consistent to the structure and constraint knowledge given in by the first view. Or, if there is an inconsistency between both views, the architect gets an immediate warning.

The two-level view approach of the prototype supports *different roles* in architectural design. Whereas defining conceptual knowledge is the task of an expert in conceptual design, the room instance view supports an architect obeying this conceptual knowledge. Of course, an advanced architect can deal with both views.

This approach has many advantages and a few drawbacks. The main advantage is that we gain *flexibility*. Defining a new room type like `bicycle garage` is done by simply creating a new node, by invoking the corresponding command of the room type view. Similarly, enforcing access

to rooms of that type from the outside is just done by creating a `obligatory` edge of type `access` between the corresponding two nodes.

This allows the advanced architect to *create* his *personal* conceptual *knowledge* or to modify that which is initially provided. It also allows us to experiment with different design concepts and their impact on created designs which is important for the explorative stage of our project.

With the analyses specified in PROGRES we are forced to *exactly define* the *consistency* relation between room type graph and room instance graph. This helps in understanding and, consequently, implementing this relation.

By modifying the consistency relation we can allow room instance graphs that *deviate* from the *conceptual knowledge* defined in a room type graph. This allows for flexible editing commands during conceptual design. Later on, new conceptual design *features* can be *inferred* from elaborated conceptual designs. This is similar to the proceeding taken in a dissertation in the field of process modeling [31]. This flexibility that conceptual design can deviate from the formulated knowledge cannot be provided with the approach presented in the last section.

There are also some disadvantages. We do not have all the *expressiveness* provided by the PROGRES language to describe conceptual knowledge. For example, complex transactions, paths, and attribute evaluation is not at hand. Of course, one could reimplement these language features by introducing the corresponding concept on the room type graph level. This, however, means to reimplement a big part of the PROGRES system.

3.2. Semantical tool support on top of ArchiCAD

As described above, the aim of *CAD systems* is to support constructive design. Positioning wall structures with doors and windows is already well supported. The 2D- and 3D-visualization features are satisfactory. So, nothing has to be done for *constructive design*. However, the semantics of constructive elements and relations between such elements cannot be defined.

The aim of the second part of our approach is to enable the architect to use a conventional tool (constructive design) but to *think* in *conceptual terms* (conceptual design). An extension of the commercial CAD system ArchiCAD has been developed, allowing the architect to use conceptual elements instead of constructive elements.

For that, the *concept* of *rooms* has been added to ArchiCAD. This is realized by an additional constructive element which we call room object. Architects think in rooms. In addition, the architect now specifies *room relations*, like the *access* between two rooms, by inserting a room link between both concerned room objects.

A room can be realized by four walls. It may also have an open side. Access between two rooms can be realized by a door between them but also by an open “wall” in between. So, we have introduced *one level* of concepts by the ArchiCAD extension. Further extensions may introduce areas of rooms, apartments on floors etc.

Inserting rooms and defining relations between rooms is, by a closer look, nothing else then the *elaboration* of the *room instance graph* consisting of typed nodes and relations between them. The architect elaborates that graph, without realizing that he uses graph technology.

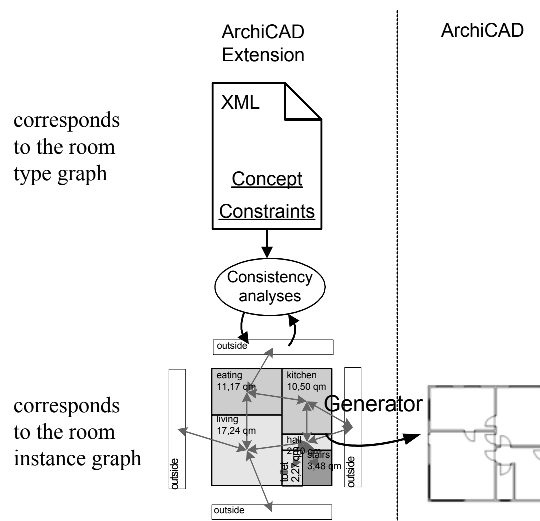


Fig. 8: Semantical extension of ArchiCAD

Analyses and consistency checks can now be performed using this graph structure. The architect uses tools realized by the theoretical background of graph technology which is completely hidden for him.

In Fig. 8 the left lower part corresponds to the *room instance graph*. This graph is realized using the *extension* features of ArchiCAD. So, the structure of room instance graph is re-implemented using corresponding system features. Analogously, we have to define the conceptual knowledge necessary for the ArchiCAD extension. This is described in an *XML file*, defining room types and requesting or forbidding relations between objects of these types. This file, therefore, corresponds to the *room type graph* introduced above. The conceptual knowledge defined in this file is used for consistency checks in the same manner as explained above. So, the situation is analogous, the way of realizing it is different. The generator to be seen on the right side of Fig. 8 creates an initial a floor plan which immediately can be shown to a customer.

As *additional information*, the geometrical positions and the dimensions of rooms are also available in a sketch as they are delivered by ArchiCAD. With the aid of this information, more complex and *powerful analyses* can be offered. These analyses are done in two steps. The first step checks if the obligatory relations between the *room objects* have been established, e. g. if whether the obligatory relation *access* between *kitchen* and *eating* is defined by a *room link*. In the second step, the geometric data are used to check whether the current sketch corresponds to the defined relations. If an access relation between two rooms has been defined, the rooms must be adjacent.

3.3. The overall solution

There are *different ways* to integrate both prototypes to form an integrated overall solution.

Up to now both *parts* of our *approach* explained in the previous two subsections seem to be strictly separated. However, they form two ends of a complete *overall solution*. This solution is not available yet.

Both parts are for different *users* with *different ambitions* and *knowledge*. The upper part, already explained, delivers tools for knowledge engineering

and pure conceptual design. The lower part allows for constructive design using conceptual knowledge (in the current state only rooms and connections). As in the lower part conceptual knowledge is also separated from the tools elaborating the design this solution is also flexible to feed in modifications or extensions of conceptual knowledge.

Looking on the software solution both parts are *similar*. We have data structures reflecting a design and data structure incorporating the corresponding knowledge. In between in both cases we find the analyses for checking the consistency between both. The realization techniques of course are *different*.

The similarity of both approaches is the *basis* for the *integration* (see Fig. 9). There are two solutions for this integration. The first solution creates a proof of concept prototype to be used in academia. The second solution can be used in industrial practice.

The first solution consists of *both* separate *systems* to be *coupled* by corresponding interface (e. g. CORBA). In this solution conceptual design and conceptual design knowledge is on the side of a PROGRES/UPGRADE prototype. The reader may remember, that this prototype supports different views. In this solution the top-down prototype is listening to all command invocations on the ArchiCAD side in order to check whether these

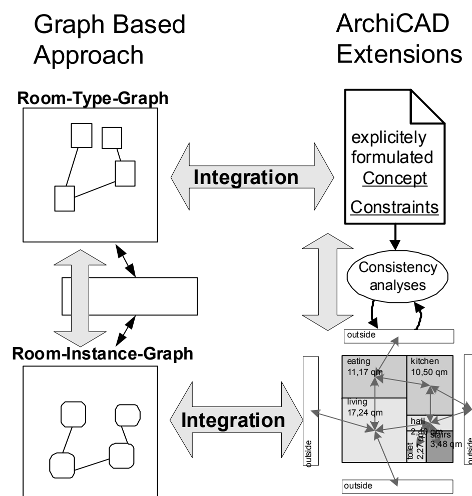


Fig. 9: Our two approaches and how they will be integrated

operations are conceptually sound. The soundness is checked between the room instance graph and the room type graph by the corresponding analyses. In this solution only the lower connection of Fig. 9 is used between both sides.

The *advantage* of this solution is its *flexibility*. We can experiment with diverse conceptual items corresponding to the dimensions of Fig. 1, thereby finding out which conceptual knowledge and which commands for conceptual design are useful for the practical life of an architect. The *disadvantage* of this solution is that such a coupled system can hardly be used in industrial practice. The argument is its *complexity*, as sketched above (infrastructure of this two level top-down prototype, time inefficiency of the coupling solution).

If experimentation with the first solution has delivered some practical results, we can *"download"* the conceptual knowledge and the tool behavior to the commercial tool ArchiCAD. In this second solution we have to encode the specifications of the first solution by *extending* ArchiCAD, in the same way as already mentioned in the last subsection. This solution yields an *industrial tool* to be used in practice. However, there is some *effort* implementing the functionality of the left hand side of Fig. 9 within ArchiCAD. The specifications written there can be used as "templates" for the "hard-wired" solution making use of the ArchiCAD extension interface. Regarding this solution both connections of Fig. 9 are only "conceptually" used.

There may be a third solution *converting conceptual knowledge* of the room type graph automatically into an XML file on the ArchiCAD side. In this case the conceptual knowledge view of the top-down prototype has the role of inputting and maintaining conceptual knowledge, whereas its use is on the ArchiCAD side. It is up to further investigations, whether this solution can be realized and, if this is the case, how complicated it is and which restrictions it induces.

In the following two sections of this paper we describe the graph-based prototype according to the top-down approach of section 3.1 and the ArchiCAD extension as described in 3.2.

4. THE TOP-DOWN SPECIFICATION-BASED PROTOTYPE

Conceptual design of buildings means to describe functional entities and their relations. This *abstraction* allows to elaborate a sketch *without* considering any *geometrical data*. Therefore, the sketch can be described as an attributed graph, where the nodes are representing rooms of the building and the edges are used to describe the relations between these rooms.

To achieve corresponding support for conceptual design by using graph technology, we use, as already described, *two graphs*. The room *type* graph describes the conceptual knowledge, the room *instance* graph the current state of a conceptual design.

In the following, these two graphs are described as well as the specification of an analysis between both levels. Furthermore, screenshots of the mechanically derived, two-view prototype are presented.

4.1. Defining conceptual knowledge by a room type graph

The room type graph contains the used room types like kitchen, toilet, or living room. In this graph, also the relations between room types are modelled. A room type graph incorporates the conceptual *knowledge* of a specific *type* of *building*.

Each room type is represented by a *graph node type*, the minimal and maximal number of allowed instances is stored as an attribute of the node. To specify relations between room types, two different edge types are available, namely to express an *obligatory* or a *forbidden* relation. The type of a relation, as e. g. `access`, is stored as an attribute of the edge. Between two node types there may be an obligatory connection of exactly one of different relation types. This is denoted by a *function* connection between two edge types.

In Fig. 10 a portion of the room type graph for our running *example* is shown. It deals with the ground floor of a single-family house. As described above, the nodes represent the room types and the edges describe relation types between them. To demand access between the room types hall, kitchen, bathroom and living an *obligatory link* between these room types is

established with the attribute `access`. To demand the room type kitchen to have a window, an obligatory link between the room type outside and the room type kitchen with attribute `view` has been installed. Access between the kitchen and the bathroom is not desired, this relation is expressed by a *forbidden link* between these room types. Between an `eating` room and a `kitchen` there may be either a *`contains`*- or an *`access`*-relation. The first is expressing that the kitchen contains a separate section, the second that both rooms are connected. The two obligatory edges are, therefore, connected by an XOR function.

For each specific *type* of a *building*, like a single-family house or a tower block, a separate *room type graph* has to be developed. Once completed, this room type graph can be used for any project for a corresponding building. It just represents the underlying concepts of this type of building.

As PROGRES does not support attributed edges, the obligatory and forbidden relations are represented by nodes with adjacent relations. In Fig. 11 the *PROGRES graph schema* of the room type graph is shown. The node

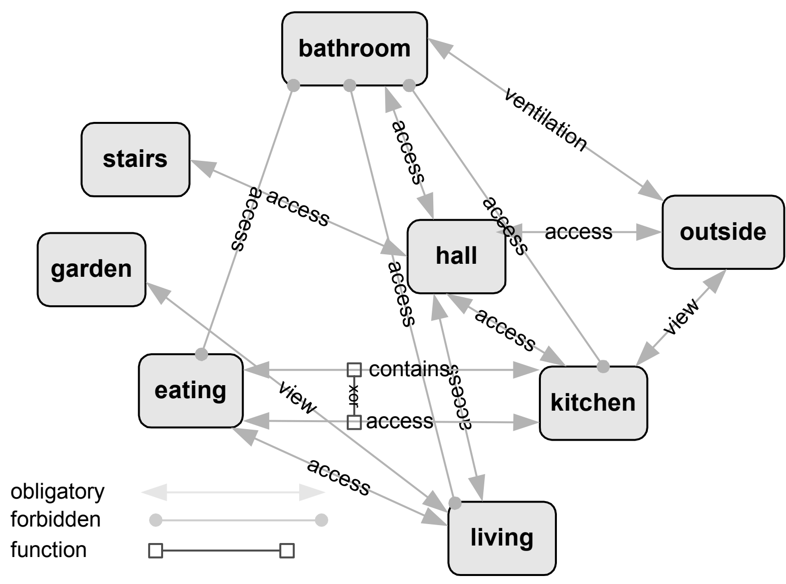


Fig. 10: Room type graph of a single-family house

classes are displayed as rectangles, the node types as rectangles with rounded corners. PROGRES edges are displayed as arrows.

The node type *t_obl_REL* describes, as instance of the node class t_RELATTR, an obligatory relation. The node type t_forbid_REL forbids a relation between two rooms. The kind of a relation, like *access,* is stored as an attribute of the t_RELATTR nodes. The node class t_FUNCTION is used to specify a function between relations. The derived node type t_XOR_FUNC describes that only one of the involved relations must exist.

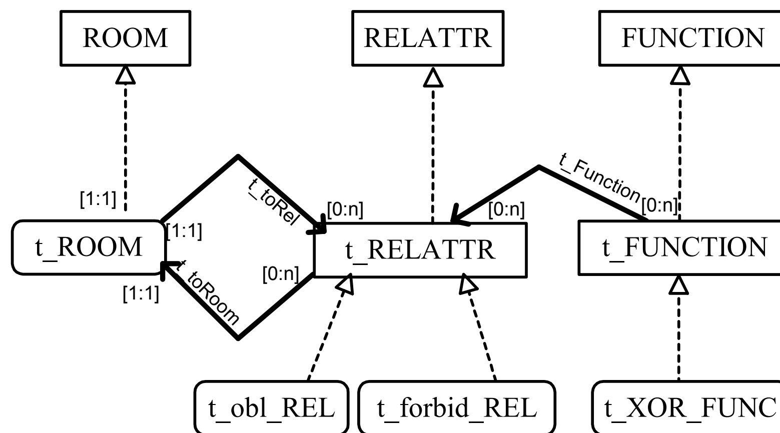


Fig. 11: PROGRES schema of the room type graph of Fig. 10

4.2. Sketching with instance graphs

In the room type graph the conceptual knowledge for specific buildings is modelled, i.e. structures and constraints. The *conceptual design* of a specific house is reflected by a *room instance graph*.

In our example, most of the room types just have *one instance*, e. g. the living room. Some room types, like bedrooms for children, have more instances. Trivially, in bigger buildings multiple occurrences come up.

In Fig. 12 a room *instance graph* is shown which is *consistent* with the room type graph displayed in Fig. 10. In the room instance graph, an edge does not demand or prohibit a relation between two rooms (as in the room type graph), it just *states* the existence of a relation between two rooms. In

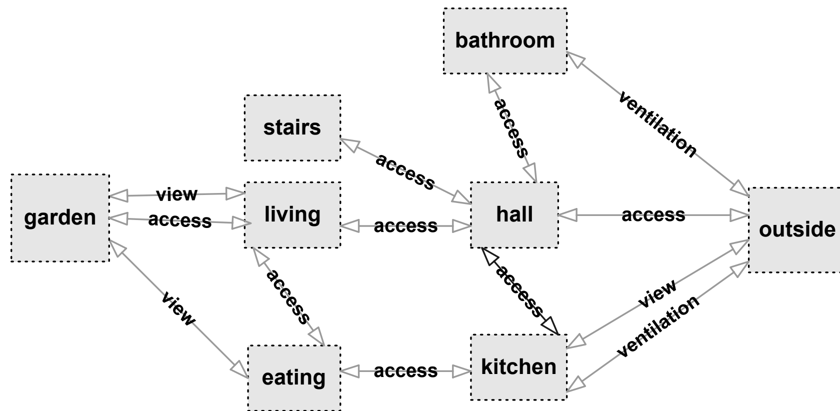


Fig. 12: Room instance graph corresponding to the room type graph

Fig. 12 there is an edge with the label *access* between the node hall and kitchen to describe that the building will have a door or some other kind of connection between both rooms. From the living room, there is access and view to the garden, in the building there will be a door and a window. As the room instance graph does not consider any geometrical data and no room dimensions, the developed sketch may serve as a model for several buildings. These buildings may look quite different, their functionality is the same.

Fig. 13 depicts the PROGRES *graph schema* for room *instance graphs*. Analogous to the room type graph, the relations between the room nodes are specified as a node class, a relation is defined by connecting two nodes of type a_ROOM with a node of type a_exists_REL, using the edge types a_toRel and a_toRoom. The type of the relation is stored in the attribute of the a_exists_REL node. An extension to the room type graph is the node class NOTIFY, which is used to handle error messages, generated by analyses.

4.3. Analyses between room type and instance graph

With the room type graph and the room instance graph the underlying concepts and a current design are described. The *consistency* between both

is checked by an *analysis*. If an inconsistency is detected, the analysis throws a notification. Depending on the priority of the violated part of the conceptual knowledge, the system displays a warning or an error messages. Both are connected to those parts of the room instance graph violating the consistency.

This analysis has to be specified. Fig. 14 shows one of the productions used for the analysis. Although it looks rather complex it is easy to understand. At the right side of the *graph pattern* (left hand side of the rule) a cutout of the room *type graph* is displayed. Node `` `2 `` and `` `4 `` are room type nodes. Node `` `5 `` represents an obligatory relation between these room types. According to node `` `7 ``, no function may operate on this relation. This pattern of three nodes of the room type graph *requests* a *relation* between any two nodes of corresponding type in the room instance graph. The type of this relation is also regarded, which is not shown here.

On the left side of the pattern node `` `1 `` and `` `3 `` represent nodes in the room *instance graph*. They are related to the room type nodes by the relation `ThisRoomsType` connecting every room instance with its room type. This pattern, therefore, *finds two nodes* of the room instance graph with room types corresponding to the room type graph. Additionally, the two room instances have *no relation* of the requested relation type.

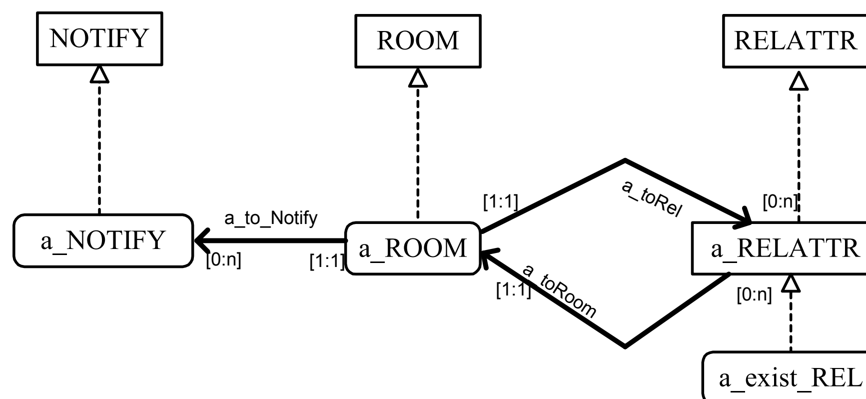


Fig. 13: PROGRES schema of the room instance graph of Fig. 12

Node `6 now claims that no *notification/warning* about this specific inconsistency is already existent. The notification node is therefore restricted: It must relate to the type of the relation and to the two node types, already mentioned.

If the described pattern has a match in the graph, *new notifications* are

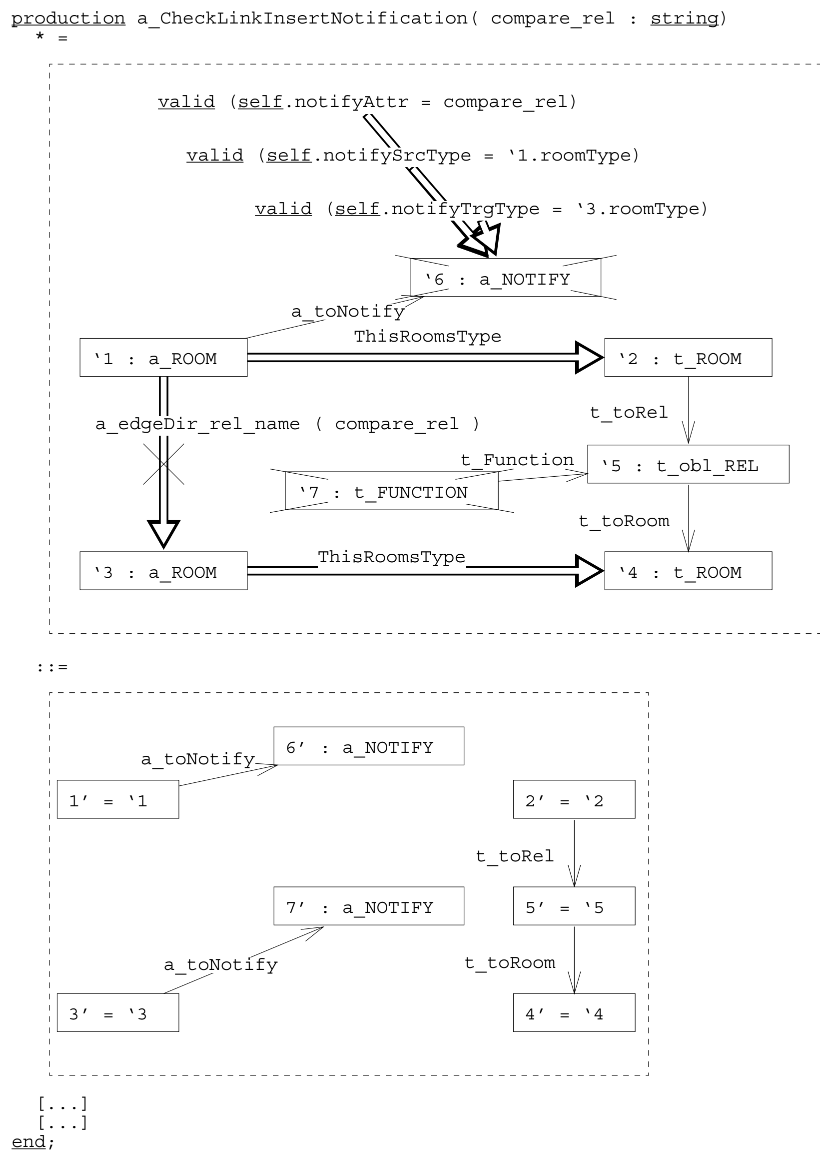


Fig. 14: One of the analysis productions checking demanded relations

inserted. This is shown in the *right hand side* of the production. Nodes `1'`, `2'`, `3'`, `4'`, and `5'` are identical replacements of the corresponding nodes to be found in the left hand side. Nodes `6'` and `7'` are new notifications, each of them linked to one of the room instance nodes that lack the requested relation.

4.4. Mechanically derived Graph Based Prototype

To experiment with and to evaluate the introduced conceptual structures and corresponding design commands, we have built a prototype with the functionality as described in 3.1 and thereby using the machinery as explained in 2.2. The *prototype* is used to model *both graphs* by taking the corresponding view.
As both graphs are inside the prototype, *consistency checks* can be easily carried out, analysing whether a building (room instance graph) corresponds to the underlying knowledge (room type graph). If the building does not correspond to the knowledge, e. g. because an obligatory link has not been established, this is shown by an error message. If the missing link is added, the error message automatically disappears.

In Fig. 15 the two views of this prototype are displayed. On the right side the *view* of the room *type graph* is shown, presenting a simple example with room types living, eating etc. and the obligatory relation access and the prohibited relation access (noaccess). From the menu the graph transformations specified in PROGRES can be invoked to build up and change the room type graph. Visualization and graph layout are completely done by the UPGRADE framework.

One the left side the *view* of the room *instance graph* is shown, some rooms are already inserted. The access relation has been installed between several rooms, but the obligatory access relation between living room and hall is still missing. The absence of this relation is displayed through an error message connected to each involved node. To correct this inconsistency, the transaction to install a link between these room instances is about to be executed, the necessary parameters are already selected.

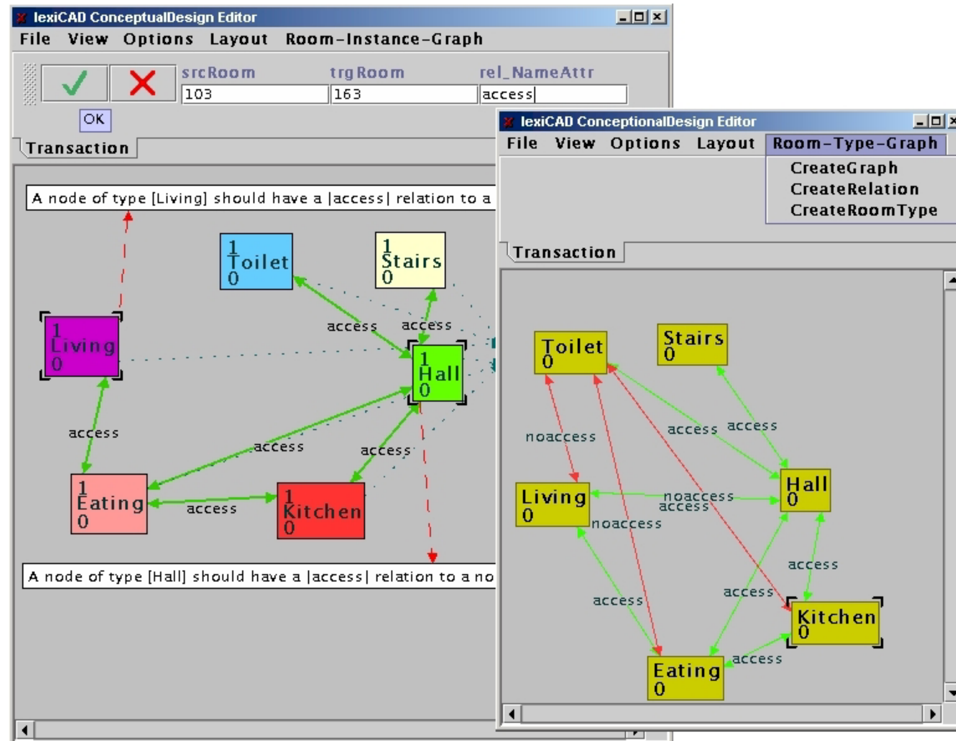


Fig. 15: The two-view conceptual design prototype showing a simple example

5. THE BOTTOM-UP ARCHICAD EXTENSION PROTOTYPE

In this subsection the *bottom-up prototype*, i.e. the extension of the commercial architecture CAD-program ArchiCAD, is described. Analogous to above, rooms are represented by nodes, the relations between these rooms are represented by edges. As graphs, their changes, as well the definition of underlying knowledge are implemented inside ArchiCAD, the architect need not understand the theoretical background. He just uses the tool invented for constructive design now having *features* for *conceptual* design.

The architect need not think about walls, windows, or doors. He can place *rooms* and define intended *relations* between rooms. We analyse geometrical data of ArchiCAD to *check* constructive room placement against the intended *underlying concepts*. We also check the design against

well-known design rules or law restrictions. To bridge the gap between room placement and constructive wall design, a *WallGenerator* creates a simple floor plan.

5.1. Structural Extensions to ArchiCAD

ArchiCAD does not offer the possibility to sketch a floor plan using rooms. The floor plan is usually constructed using the wall tool. As architects rather think in rooms during conceptual design, the room concept was added to ArchiCAD. The *room object* is an additional object inside ArchiCAD. In the 2D-view a room object is drawn as an rectangle with the room type, the area of the current instance being displayed inside the room. Each room type (kitchen etc.) has an unique color to be distinguish from others with different functionality. The architect can insert, drag, and resize a room object as he does with any other ArchiCAD object. The 3D-view offers an impression of room volumes.

To define relations between rooms, our extension offers a feature to install links connecting two room objects. These *room links* can have different attributes, like *access*, *ventilation,* or *view*. As the architect sketches the floor plan and not the underlying concepts, the edge type describes the existence of such a relation and not the obligation or prohibition. A feature called the `EditorHelper` assists in moving and

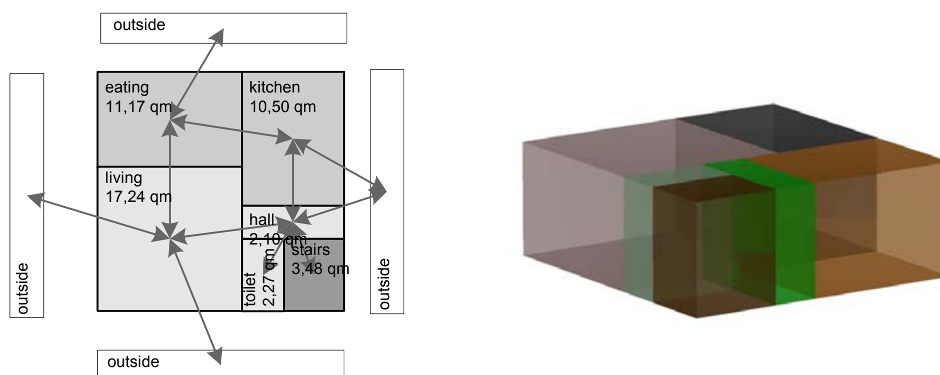


Fig. 16: The floor plan enriched by room objects and relations, 3D-view of the floor

resizing rooms by adjusting connected rooms after a resize operation.

By sketching the floor plan using room objects and defining relations between them, the architect does not just design a floor of a building. Without being aware, he *models* a graph structure, namely the room *instance graph*. As additional information, the geometrical data complete this graph to form a concrete sketch.

5.2. Analysis of a sketch

The *ConstraintChecker* permanently *supervises* that none of the defined knowledge rules are violated. It reacts on notifications sent by ArchiCAD, if an object is created, deleted, or modified. If one of the corresponding actions did *violate* a *rule*, an error message is displayed to inform the architect about the problem (see Fig. 17).

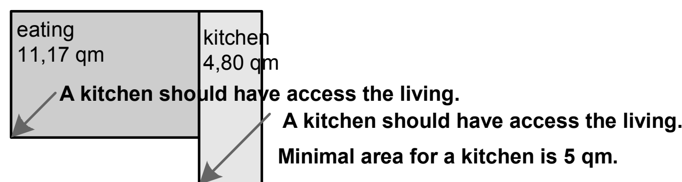


Fig. 17: Error messages of the ArchiCAD extension

The *information* of the *graph structure* can be used to perform these consistency checks. Moreover, the *geometrical data* also available allow to perform more powerful analyses. So, not only the absence of an obligatory link can be found as design error (due to graph information). Furthermore, also errors, like too small room dimensions can be discovered (due to geometrical data).

5.3. Connection to later phases of design

When the conceptual layout of a floor plan is finished, the walls have to be generated. This need not be done by hand as if the architect would use standard ArchiCAD. Instead, with the information of room objects and

room links a new tool `WallGenerator` constructs an *initial wall structure* for a floor plan.

Starting from this initial wall structure, as displayed in Fig. 18, the architect now *continues* working in a traditional way. He elaborates and details the *constructive design*.

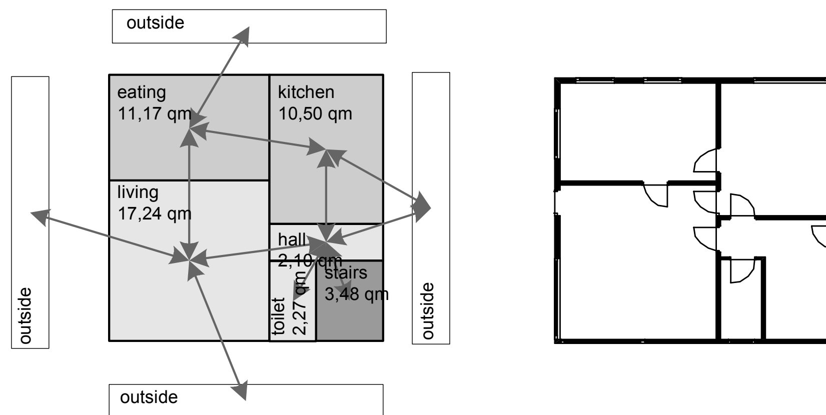


Fig. 18: Floor plan with an generated, initial wall structure

6. FUTURE PLANS

In the future, we plan to extend this project in various ways. The graph-based prototype and the extension prototype will *merge in a single approach*. In a first step the graph-based prototype will be used as an editor for the room type graph which is used to analyze the graph of room objects and room links of the ArchiCAD extension. A second step will reconstruct the room instance graph from the ArchiCAD extension model and directly analyze it by code generated from the graph rewriting system within the graph-based prototype. Here, knowledge from the CHASID project about prototype integration can be reused. Other steps, as sketched in 3.3 will follow.

The following plans are on the side of the ArchiCAD extension. The links between room objects will have further influence on the geometry of

the room objects. An *automatic layout* will propose initial positions and sizes for room objects. Repositioning a room object might then have effects on other floors or on room objects further away.

To create a better integration of constructive design and conceptual design, the wall generator will be extended to a *wall integrator*. Changes in wall placement can then cause the geometry of room objects to change and initiate link creation. We will use our knowledge on incremental integration tools to create a reactive integrator.

The following plans apply to both prototypes. To evaluate our approach we will elaborate *bigger examples* that having extended functional needs. We have done first steps by modeling the overall structure of an airport and a workshop.

To cope with different granularities in the structure of large, *complex* buildings we will introduce a *hierarchy of objects*. When laying out an airport, an area for the canteen must be reserved. The internal layout of that area is not of interest when relating it to the entrance area or check-in counters. Yet the canteen itself is complex enough to make it worthwhile to plan, check, and analyze the arrangement of e. g. serving area, cash desk, passage ways, and scullery.

Acknowledgement: The authors are indebted to A. v. Humboldt-Stiftung for a grant allowing the cooperation mentioned in 1.3. The partners of the project have contributed to many fruitful discussions.

REFERENCES

[1] Al-Shihi, B., Chung, P., and Holdich, R.: *A Decision Support Tool for the Conceptual Design of De-oiling Systems.* In: Logananthraj, R., Palm, G., and Ali, M.: Intelligent Problem Solving, LNAI 1821, pages 334-344, Springer, Heidelberg (2000)

[2] Alexander, C.: *Eine Mustersprache.* Löcker, (1995)

[3] Becker, S., Haase, T., Wilhelms, J., and Westfechtel, B.: *Integration Tools Supporting Cooperative Development Processes in Chemical Engineering.* In: Proc. 6th World Conference on Integrated Design Process Technology, Pasadena (2002)

[4] Borkowski, A. and Szuba, J.: *Graph Transformation in Architectural Design.* In: Computer Assisted Mechanics and Engineering Science, Vol. 3, pages 109-119 (2001)

[5] Böhlen, B.: *Basisschicht eines Rahmenwerks für graphbasierte Anwendungen*, Diploma Thesis, RWTH Aachen, Aachen, Germany (1999)

[6] Chen, Y. Z. and Maver, T.: *The Design and Implementation of a Virtual Studio Environment.* In: Proc. 2. East-West Conference on IT in Design, Moscow (1996)

[7] Cichocki, P., Gil, M., and Pokojski, J.: *Heating System Design Support.* In: Ian Smith: Artificial Intelligence in Structural Engineering, LNAI 1454, pages 240-248, Springer, Heidelberg (1998)

[8] Cremer, K.: *Anwendung von Graphentechnik zum Reverse Engineering und Reengineering*, Ph. D. Thesis, RWTH Aachen, Deutscher Universitätsverlag, Wiesbaden (1999)

[9] Eich, R.: *Honorarordnung für Architekten.* Werner, (1996)

[10] Flemming, U.: *Case-Based Design in the SEED System.* In: Knowledge Based Computer Added Architectural Design, pages 69-91, Elsevier, New York (1994)

[11] Gatzemeier, F.: *Patterns, Schemata, and Types - Author Support Through Formalized Experience.* In: ICCS 2000, International Conference on Conceptual Structures, LNAI 1867, Springer, Heidelberg (2000)

[12] Gips, J.: *Computer Implementation of Shape Grammars.* In: Workshop on Shape Computation, MIT, Massachusetts (1999)

[13] Gips, J. and Stiny, G.: *Shape Grammars and the Generative Specification of Painting and Sculpture.* In: Freiman, C. V.: Proceedings of IFIP Congress 71, pages 1460-1465 (1972)

[14] Göttler, H., Günther, J., and Nieskens, G.: *Use Graph Grammers to Design CAD-Systems.* In: Rozenberg, G.: Graph Grammers and Their Application to Computer Science, Lecture Notes in Computer Science 532, pages 396-409, Springer (1990)

[15] Herzberg, D. and Marburger, A.: *E-CARES Research Project: Understanding Complex Legacy Telecommunication Systems.* In: Sousa, P. and Ebert, J.: Proceedings of the 5th European Conference on Software Maintainance and Reengineering (CSMR), pages 139-147, IEEE Computer Society Press, Los Alamitos, Ca. (2001)

[16] Jäger, D.: *Generating Tools from Graph-Based Specifications.* In: Information and Software Technology, Vol. 42, pages 129-139 (2000)

[17] Jäger, D., Schleicher, A., and Westfechtel, B.: *AHEAD: A Graph-Based System for Modeling and Managing Development Processes.* In: Nagl, M., Schürr, A., and Münch, M.: Proc. Workshop on Applications of Graph Transformations with Industrial Relevance, LNCS 1779, pages 325-340, Springer, Heidelberg (1999)

[18] Kiesel, N., Schürr, A., and Westfechtel, B.: *GRAS, A Graph-oriented Software Engineering Database System.* In: Information Systems, Vol. 20(1), pages 21-51 (1995)

[19] Landzettel, R. and Schwier, V.: *Wohnhaustypen für die Selbsthilfe*, AVA-Arbeitsgemeinschaft zur Verbesserung der Agrarstruktur in Hessen e.V. (1972)

[20] Lefering, M.: *Integrationswerkzeuge in einer Softwareentwicklungs-Umgebung*, Ph. D. Thesis, RWTH Aachen, Shaker, Aachen (1994)

[21] Maver, T.: *A number is worth a thousand pictures.* In: Automation in Construction, Vol. 9, pages 333-336 (2000)

[22] Menal, J., Moyes, A., McArthur, S., Steele, J. A., and McDonald, J.: *Gas circulator design advisory system: A web based dicision support system for the nuclear industry.* In: Loganantharaj, R., Palm, G., and Ali, M.: Intelligent Problem Solving, LNAI 1821, Springer, New York (2000)

[23] Meyer, O., Gatzemeier, F., Fuß, C., and Kirchhof, M.: *Inferring Structure Information from Typography.* In: Digital Documents and Electronic Publishing (DDEP00), LNCS 1923, Springer (2000)

[24] Nagl, M.: *Building Tightly Integrated Software Development Environments: The IPSEN Approach.* In: LNCS 1170, Springer, Heidelberg (1996)

[25] Nagl, M., and Westfechtel, B.: *Integration von Entwicklungssystemen in Ingenieuranwendungen.* Springer, Heidelberg (1998)

[26] Neufert, E.: *Bauentwurfslehre.* Vieweg, Wiesbaden (2000)

[27] Ngo, D. Q. and Rüppel, U.: *BINAS - Ein Entscheidungshilfesystem für die Bestandsanalyse von Bauwerken.* In: Int. Kol. über die Anwendungen der Informatik und der Mathematik in Architektur und Bauwesen (ikm), Weimar (1997)

[28] Pisthol, W.: *Handbuch der Gebäudetechnik.* WERNER, Düsseldorf (1999)

[29] Rüppel, U.: *Ganzheitliches Managment von Bauprojekten aus Auftraggebersicht auf der Basis integrierter Ablauf- und Kostensteuerung.* In: Bauingenieur, Vol. 3, pages 105-110 (1998)

[30] Rüppel, U., Diaz, J., and Meißner, U.: *Entscheidungsunterstützung geotechnischer Planung, Konstruktion und Steuerung mit objektorientierten Baugrundmodellen.* In: Bautechnik, Vol. 9, pages 595-604 (1996)

[31] Schleicher, A.: *Roundtrip Process Evolution Support in a Wide Spectrum Process Management System*, Ph. D. Thesis, RWTH Aachen, DUV, Wiesbaden (2002)

[32] Schürr, A., Winter, A. J., and Zündorf, A.: *The PROGRES approach: Language and environment.* In: Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G.: Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools 2, pages 487-550, World Scientific Publishing Company, Singapore (1999)

[33] Shaviv, E. and Peleg, U.: *An integreted KB-CAAD System for the Cesign of Solar and Low Energy Buildings.* In: Schmitt, G.: CAAD Futures' 91 (1991)

[34] Szuba, J., Grabska, E., and Borkowski, A.: *Graph Visualisation in ArchiCAD.* In: Nagl, M., Schürr, A., and Münch, M.: Application of Graph Transformation with Industrial Relevance, LNCS 1779, Springer, Heidelberg (1999)

[35] Wall, R. and Seidle, R.: *CastleMaker.* In: FAXMAX, Excursions on Density, pages 231-247 (1997)

[36] Westfechtel, B.: *Models and Tools for Managing Development Processes.* In: LNCS 1646, Springer, Heidelberg (1999)

[37] Whitehead, B. and El'Dars, M. Z.: *The planning of single storey layouts.* In: Building Science, Vol. 127 (1965)