Available online at www.sciencedirect.com
SCIENCE @ DIRECT®

ELSEVIER Electronic Notes in Theoretical Computer Science 127 (2005) 141–156

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

New Object-Oriented PROGRES for Specifying the Conceptual Design Tool GraCAD

Janusz Szuba [1]

Real-Time Systems Lab
Darmstadt University of Technology
Merckstr. 25, D-64283 Darmstadt, Germany
Janusz.Szuba@es.tu-darmstadt.de

Abstract

This paper deals with the application of graph transformations for the specification of conceptual design tools. We show how the graph rewriting system PROGRES is used for specifying the graph part of the conceptual method for architects in which functional requirements of the building to be designed are elicited by means of graph structures. The consistency of the specified requirements and whether a design matches those requirements is verified with graph constraint checkers. We consider how the new object-oriented extensions of the PROGRES language, i.e. packages and node objects with redefinable methods can be used to achieve the required constraint monitoring and preserving functions in the form of graph checker objects. The prototype for our method, called GraCAD, is created with UPGRADE - the recently developed Java framework for developing visual applications based on a PROGRES specification, and the commercial system for architects ArchiCAD.

Keywords: PROGRES, CAD, ArchiCAD, Conceptual Design

1 Introduction

Designers, especially architects, very frequently use graph structures to represent the functional and spatial relations of the object to be designed. Based

[1] Many thanks to Andy Schürr for fruitful discussions and assistance in the paper preparation, and to the PROGRES and UPGRADE team for its support (in particular Bodo Kraft, Galina Volkova, and Marita Breuer). This research was supported by European Research Training Network "SegraVis" and Computational Engineering Center of TU Darmstadt.

1571-0661 © 2005 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2004.12.032

on this observation, a new conceptual design method for buildings has been created in which the functional requirements and constraints of the building to be designed are specified by using graph structures. The consistency of the specified requirements and whether the design meets the requirements is verified by means of graph checkers. For prototyping the graph part of our method we use the graph-rewriting system PROGRES [21] developed at the RWTH Aachen, i.e. we use a kind of graph transformation-based approach for knowledge representation purposes. The design tool GraCAD, which utilises this method, can be seen as a conceptual pre-processor for a new generation of CAD-tools. GraCAD was created with the usage of PROGRES UPGRADE Framework [2] and the CAD design tool for architects ArchiCAD [1].

In [23] we used the PROGRES mechanisms of *constraints* and *repair actions* for specifying the constraint checks for the GraCAD method. However, these mechanisms turned out to be still difficult to use. Moreover, the old PROGRES specifications (i.e. before ver. 11) were not easily extendable and adaptable to the new types of building and constraints. Therefore, we have decided to avoid using constraints and repair action and use the new PROGRES constructs of *packages* and *methods*, which allows for the creation of more modular and clearer specifications. The main topic of this paper is the usage of these new object-oriented features of PROGRES for specifying GraCAD and presenting the new object-oriented style of specifying graph-based tools.

The joint Polish-German research project "Graph-based tools for conceptual design in Civil Engineering" was the starting point for our research. This project has now been completed, however the research is continued at the RWTH Aachen in the direction of the specification of parameterizable knowledge for buildings ([14],[15]), and at the TU Darmstadt (supported by SegraVis [1]) with a main emphasis on the integration of UML activity and use case diagrams with ArchiCAD. The results described in sections 2 and 4 can be viewed as a further development of the research reported previously in [23] and [24], i.e. combining, for the first time, graph transformation techniques with a conceptual design approach for buildings, based on functionality analysis. Section 2 gives a short description of our method. Section 3 shows how we use the recently introduced PROGRES concepts of packages and methods to specify the checkers of our method. Section 4 concerns the GraCAD prototype (realizing the method) which was created with UPGRADE, the framework for developing visual applications which use PROGRES graphs as their internal data model, and the commercial CAD tool for architects ArchiCAD. Section

[1] European Research Training Network SegraVis - Syntactic and Semantic Integration of Visual Modelling Techniques

4 discusses related work, and the last section summarizes the new PROGRES features.

2 Graphs in Supporting Conceptual Design

As mentioned previously architects very frequently use graphs to depict the functional and spatial relations of designed objects. Furthermore, they use control flow graphs, similar to UML activity diagrams, to show the order of activities performed in the considered design object; i.e. — similar to software engineers — architects follow a use case driven approach for requirements elicitation purposes (cf. [17]). Based on these observations, we have created a method that addresses the conceptual phase of architectural design in which the functional requirements and constraints for designed buildings are specified in the form of graph structures. In this method, UML use case and activity diagrams are integrated with so-called area and room graphs, which are then translated into a prototype design. One of the main advantages of the graph-based design approach introduced thereby is the possibility to specify domain-specific design rules and norms on area and room graphs in the form of constraints on a very high level, and to derive the corresponding consistency checking code automatically by using the graph transformation system PROGRES. In the following, a short description of the method is presented. Use case diagrams and area graphs are excluded from this description due to lack of space. Details can be found in [23].

First, the architect defines scenarios for the usage of the building to be designed in the form of *activity graphs* (*UML activity diagrams*). These activity graphs model the most frequent and important behaviours of the users. By creating scenarios in the form of activity graphs for various types of users, the functionality of the object is considered from various points of view (from the perspective of the client of the swimming pool, the lifeguard, etc.). Then the architect decomposes the designed object into rooms. For this purpose, he/she creates a *room graph* in which room nodes and relations between them like accessibility, visibility, and adjacency are specified. Afterwards, *activity nodes* from activity graphs are *mapped onto room nodes*. Mapping activities onto a given room node means that these activities are performed in the assigned room. In this way a functional requirements' graph for the building, consisting of rooms, activities, and edges between them is created. Functional requirements defined in this way may be checked with respect to quite a number of general or domain specific consistency rules. The PROGRES implementation of such a checker (i.e. checking whether users are able to perform the activities comfortably in the order imposed by the activity graph if the building has a

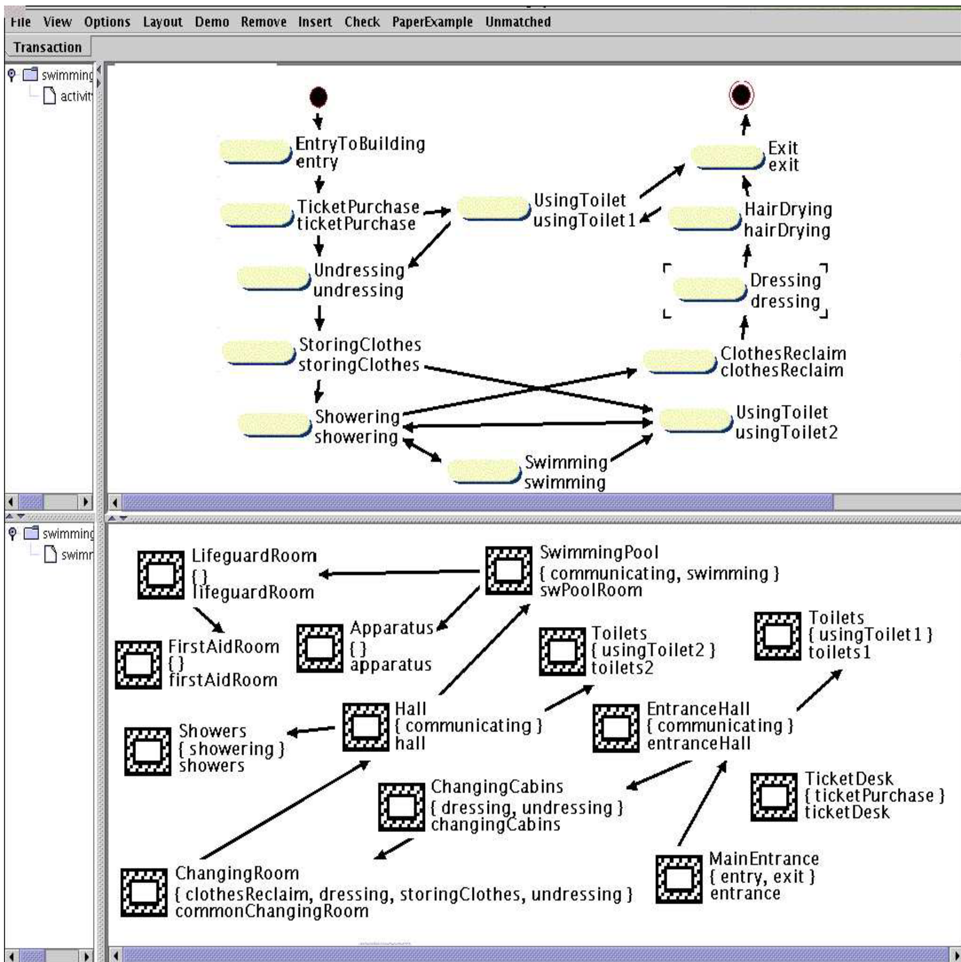

Fig. 1. The GraCAD graph editor for specifying the functional requirements of the building to be designed

structure matching the defined room graph) is presented in section 3.

Fig. 1 shows the GraCAD graph editor (based on PROGRES UPGRADE) used to specify the *activity diagrams* and *room graphs* for the designed building. The upper right window of GraCAD shows the activity (graph) diagram for a swimming pool client. The lower right window contains the room graph for a swimming pool. In the presented room graph of the swimming pool, only the room accessibility relation is shown. For every room in the room graph, the identifiers of activities attached to a given room are listed in brackets. After specifying consistent functional requirements, the architect creates a *prototype design/floor layout* of the building.

Fig. 2 shows the prototype created in the ArchiCAD environment. The rooms are marked with the ArchiCAD element *zone* and than mapped onto room nodes from the room graph. Afterwards, based on the geometrical elements of the prototype, the relations between zone elements are computed. The left hand side part of Fig. 2 shows the zone graph for the swimming pool presented on the right hand part of the picture. In the zone graph, room accessibility and room adjacency relations are displayed. Due to the mapping of zones into room nodes it is possible to verify (by checkers) whether the design matches the room structure specified in the graph of functional requirements. Details concerning the GraCAD graph editor and the ArchiCAD part of the GraCAD prototype can be found in section 4.

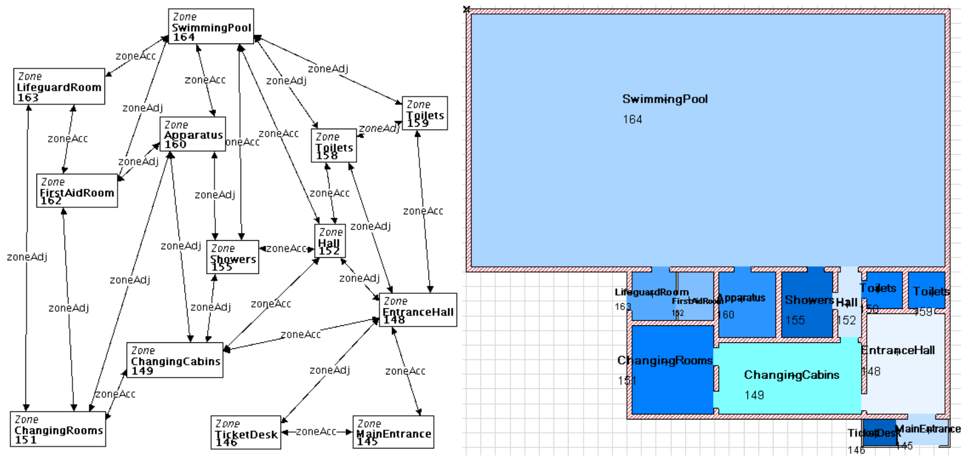

Fig. 2. The ArchiCAD swimming pool design and the PROGRES zone graph for this building computed by the GraCAD add-on

3 New PROGRES for GraCAD Checkers

For prototyping the graph part of GraCAD we use the PROGRES system. In this section we show an example of a checker implemented with new PROGRES language constructs, i.e. *packages* and *methods for node classes/node types*. These constructs were introduced to support the modularization of PROGRES specifications and simplify the reuse of specification fragments.

On the basis of the *UML package* concept, a module concept for PROGRES (called *package* as well) was developed [20]. The PROGRES *packages* allow us to conceal the implementation of transformations, queries, paths or other definitions and improve the readability of the specification. Basically, a *package* represents a container for various declarations. These declarations

can have visibility flags. With the help of visibility flags every declaration can be characterised with regard to its accessibility from other packages. We distinguish between three visibility flags: *public* (marked in PROGRES as "+"), *protected* ("#") and *private* ("–"). An element marked with the visibility *public* can be accessed from any other package. Elements which are *protected* can only be accessed from specialising packages. *Private* elements cannot be accessed at all from other packages. Packages can be put into a relation to other packages, like *specialisation* and *import* relations. The *import* relationship extends the name space of the importing package. It is possible to reference declarations of other packages in the importing package as soon as they are marked as public. The *specialisation* relationship can reference the same elements as the import relationship as long as they are either marked protected or public. Furthermore, it is possible to specialise node class declarations, i. e. a node class defined in the specialising package may inherit from a node class defined in the specialised package. The import and specialisation relationship itself can also have different visibilities. Details concerning the package relation visibilities can be found in [16].

Fig. 3 shows the PROGRES package diagram and class diagram for the GraCAD graph specification. In the graph schema view mode of the PROGRES system, classes and packages are visible at the same time on the graph schema but for the sake of clarity we have divided the schema into the package diagram and the class diagram. For the same reason we displayed only selected packages, classes and relations. In the package diagram the *import* relationship is indicated by the dotted arrow between the packages and *specialization* by the solid one. In the class diagram *abstract classes* (called in PROGRES *node classes*) are marked as rectangles, *concrete classes* (called in PROGRES *node types*) as rectangles with rounded corners.

The classes displayed in Fig. 3 form three UML layers:

(i) *standard UML language layer* with the meta-model of UML activity diagrams (*umlactdiag* package)

(ii) *extended UML layer for architectural conceptual design in general.* This layer consists of an activity diagram for buildings (implemented in *buildactdiag* package) and a room graph (*roomgraph* package) which define the functionality of a given building (*building* package)

(iii) *extended UML for architectural conceptual design specific layer for a given type of building* (in our case for a swimming pool). This layer is defined in *swpoolactdiag, swpoolroomgraph, swpoolbuilding* packages.

For every layer, graph constraint checkers can be defined. The package *buildingchecker* consists of checkers for the second layer and *swpoolbuildchecker*

for the third one. In the next paragraph, the classes and packages are described in detail.

In the GraCAD specification for the swimming pool the following packages are defined:

- *basic* - consists of an abstract class *Object* which is the base class for *Activity, ActivityDiag, Room, RoomGraph, Building, Error* classes. The *Object* class has the *hasChecker* attribute which is a set of nodes of a type *Checker*. This attribute contains checkers which are appropriate for a given node/object. Those checkers are instantiated based on the meta (static) attribute *checkerTypes*, which is a set of node types derived from the type *Checker*. The attribute *checkerTypes* is redefined in the classes derived from the class *Object*. The package *basic* and the class *Object* are not shown in Fig 3.
- *checker* - consists of the abstract class *Checker* that is the base class for all checkers, and the abstract class *Error* - the base class for all classes used for marking errors by checkers.
- *umlactdiag* - contains classes and edges used for representing UML activity diagram, i.e. the abstract class *Activity*, concrete classes *Start* and *Stop* derived from *Activity*, the abstract class *ActivityDiag*, and edges: *next* between two *Activities*, *hasActivity* between *ActivityDiag* and *Activity* (indicating which activities belong to a given activity diagram).
- *roomgraph* - contains classes and edges used for representing the room structure graph of the building to be designed, i.e. *Room* (the room of the building to be designed), *RoomGraph* (the graph of rooms of the designed building) and edges: *hasRoom* between *RoomGraph* and *Room* (indicating which rooms belong to a given room graph), *roomAccess* between two *Rooms* (if between rooms r_1 and r_2 is the *roomAccess* edge it means that r_2 is directly accessible from r_1).
- *buildactdiag* - contains classes and edges for representing the activity diagram for the designed building, i.e. abstract classes: *BuildingActivity* (inherited from *Activity*) which represents the activity performed in the designed building, *BuildActDiagram* representing the activity diagram for the designed building, the concrete class *Communicating* which represents the communicating activity and the edge *performedIn* between *BuildingActivity* and *Room* (if between a building activity a and a room r is the *performedIn* edge, it means that the activity a is performed in the room r).
- *building* - consists of the *Building* class representing the designed building, the *actDiag* edge between *Building* and *BuildActDiagram* indicating which building activity diagrams belong to a given building, and *roomGraph* edge between *Building* and *RoomGraph* indicating which room graphs belong to

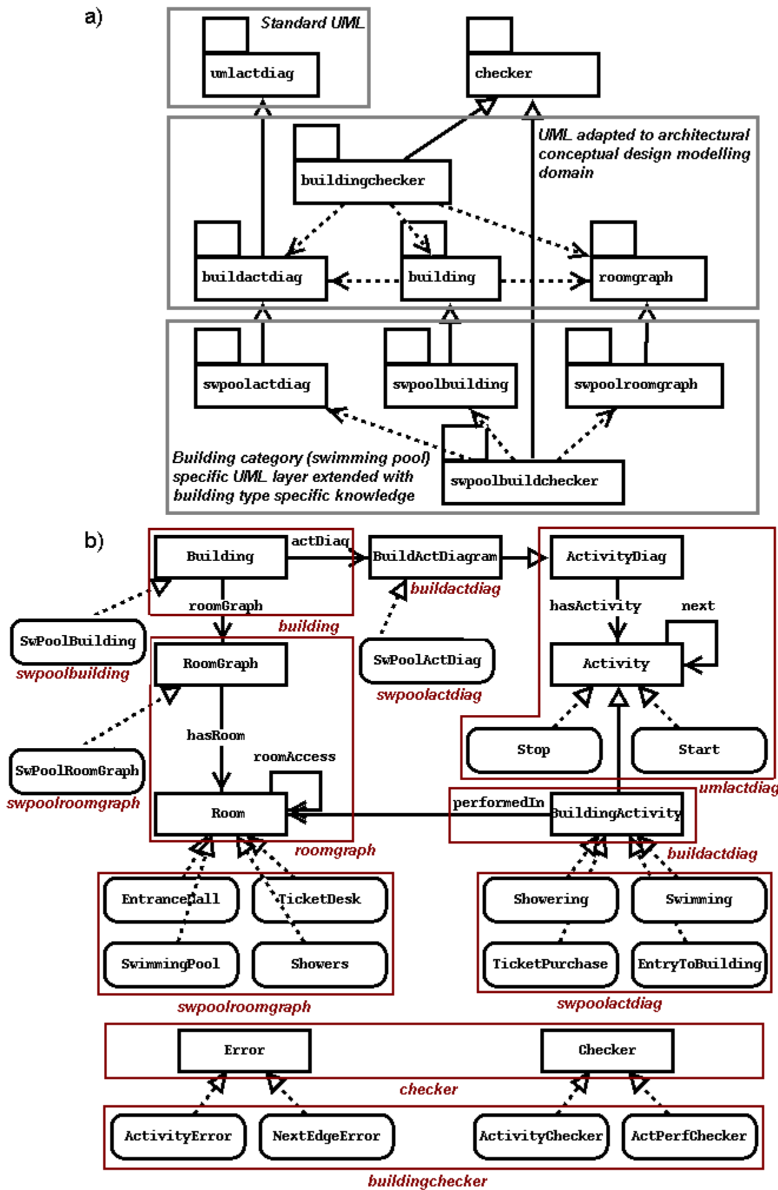

Fig. 3. GraCAD graph schema: a) packages b) classes

a given building.

- *swpoolactdiag* contains activities performed by users in a swimming pool building like *Swimming, Dressing, Undressing*, etc. (inherited from *BuildingActivity*) and *SwPoolActDiag* (activity diagram for swimming pool) inherited from *BuildActDiagram*.

- *swpoolroomgraph* contains rooms specific for a swimming pool like *Bathroom, ChangingRoom, SwimingPool*, etc. (inherited from *Room*) and *SwPoolRoomGraph* (room graph for swimming pool) inherited from *RoomGraph*.
- *swpoolbuilding* contains *SwPoolBuilding* (swimming pool building) inherited from the class *Building*.
- *buildingchecker* contains two checkers for a building activity diagram: *ActPerfChecker* (activity performance checker) checks whether users are able to perform the activities comfortably (details in the following paragraph), *ActivityChecker* which validates whether to every activity a room is assigned. *ActivityError* and *NextEdgeError* classes inherited from *Error* are used accordingly by *ActPerfChecker* and *ActivityChecker* to mark errors.

In the presented class diagram only 2 checkers are shown but others can be defined as well, for instance checking the order of activities in the activity diagram (cf. [23]).

In the following part of this section the mechanism of PROGRES methods is described and explained by the example of *ActPerfChecker*. This checker verifies whether users are able to perform the activities comfortably in the order imposed by the activity graph if the building has a structure matching the defined room graph. In other words, if the *next* edge is between two activities a_1 and a_2 then it should be possible to reach "conveniently" from every room r_1 attached to a_1, to one of the rooms attached to the activity a_2. "Conveniently" means that: *(a)* room r_1 should be one of the rooms attached to the activity a_2 or *(b)* from r_1 it should be possible to access directly one of the rooms attached to a_2 or *(c)* from r_1 it should be possible to access indirectly through the *sequence of communication rooms* one of the rooms attached to a_2 (communication room is a room with attached the activity of a type *Communicating*). Let us look at the example in Fig. 2. The activity with the identifier *swimming* can be conveniently performed after the *showering* activity because the condition *(c)* is fulfilled, i.e. from the *showers* room (attached to the *showering* activity) it is possible to access indirectly through the communication room *hall* the room *swPoolRoom* (attached to the *swimming* activity). The activity *ticketPurchase* cannot be conveniently performed after the activity *entry* because from the room *entrance* it is not possible to reach directly or indirectly (through communication rooms) any of rooms attached to the *ticketPurchase* activity, and *entrance room* is not attached to the *ticketPurchase* activity either.

Up to the PROGRES version 10.3, defining classes in the object-oriented sense as abstract data types with operations on them was not possible. The classes could have possessed only attributes but not methods. In the latest

```
node class Checker
  intrinsic
    key checkerId : string := "";
  methods
    transformation + validate;
    transformation + repairAction
    =
      skip
    end   ;
end;
```

Fig. 4. GraCAD specification - the Checker node class

PROGRES (i.e. version 11, cf. [16]), *methods* for node classes have been introduced to obtain completely featured object-oriented classes. The formal syntax of *methods* introduced in the latest PROGRES is mainly based on the syntax for transformations and queries from the previous version. A new key word *methods* (cf. Fig. 4) was introduced. After this word, the definition of methods for a given class follows. A transformation method has almost the same syntax as a "classic" transformation in PROGRES. It starts with the keyword *transformation*, followed by a visibility flag (the same as for node classes), the transformation's identifier (i. e. name), an optional formal parameter list, and a qualifier which indicates whether this transformation is specified to be partial or total and which also allows for parallel execution of transformations. Besides transformation methods, query methods are available as well. The syntax of the query methods is very similar to the syntax of transformation methods (cf. [16]). The biggest difference to "classic" transformations is the optional transformation body. For methods we have now the possibility to define abstract transformations, which can be implemented, i. e. refined, later in a subclass of the corresponding node class. Within methods it is permitted to use the keyword *self* which designates the particular instance of the node this method is applied to.

The class Checker (Fig. 4) from the GraCAD specification consists of two transformation methods *validate* and *repairAction*. The transformation *repairAction* has a defined body, the method *validate* is an abstract transformation which must be defined in *Checker*'s subclasses. Fig. 5 shows the PROGRES node type *ActPerfChecker* which is a subclass of the node class *Checker*. In the section *methods*, the private method *checkNextEdge* is defined. After the name of this transaction, the qualifier "*" follows. This leads to the application of this transaction to all its matches in parallel. In the implementation of *checkNextEdge* a PROGRES path expression *connected* defined between two *Room* classes is used (*Path* declaration defines a derived binary relationship between nodes and has the same elements as an edge type declaration.). Two rooms r_1 and r_2 are in the relation *connected* when r_1 is the same as r_2 or r_1 is connected with r_2 by the sequence of communication rooms.

The *connected* path declaration is not shown in the paper due to lack of space. The left hand side of the transaction *checkNextEdge* finds two activities (‘3, ‘4) of a type *BuildingActivitiy* connected by the *next* edge. Those activities are connected by *hasActivity* edges with a diagram of a type *BuildActDiag* which is linked to the *self* node by the *hasChecker* edge (The *self* node indicates the object for which the method is defined.). The activity ‘3 is linked by the *performedIn* edge with a room (‘6) of a type *Room*. The activity ‘3, ‘4 and the room ‘6 cannot be connected to a node of a type *NextEdgeError* adequately with edges: *activity1, activity2, room*. The room ‘6 can not be linked with the activity ‘4 by the path expression “connected & <-performedIn-”. If such a sub-graph is found, then the right hand side of the transaction is applied, i.e. a node of a type *NextEdgeError* is created and linked to the nodes ‘3, ‘6, ‘4 adequately with edges *activity1, activity2* and *room*. If we apply this rule to the inconsistency from our example, i.e. to the activities *entry* and *ticketPurchase*, and the room *entrance*, and for those nodes *NextEdgeError* has not been inserted yet, then an instance of the *NextEdgeError* node will be created and linked appropriately with *entry*, *ticketPurchase* and *entrance*.

Fig. 6 shows the last part of the considered checker, i.e. the *redefinition*

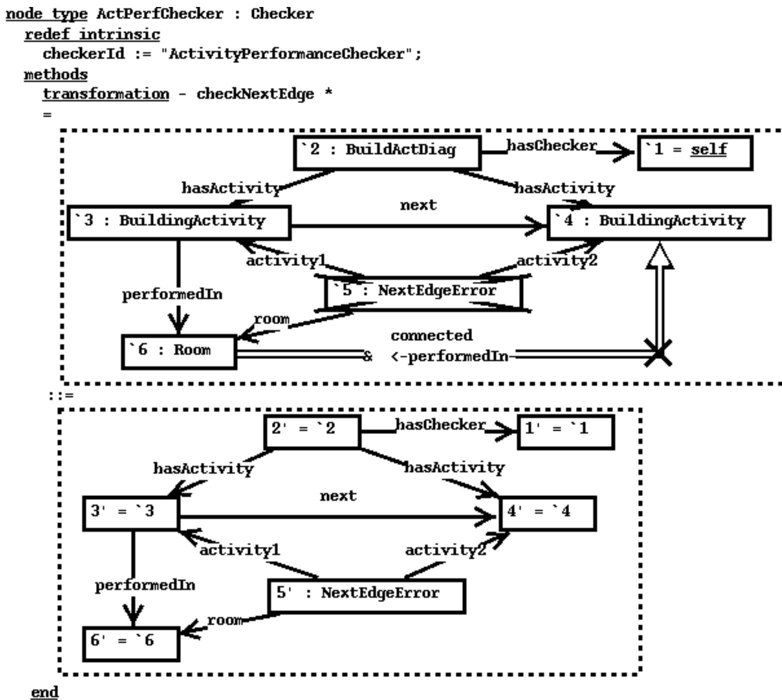

Fig. 5. The PROGRES class ActPerfChecker with the method checkNextEdge

section of the *ActPerfChecker* class. The *Redefinition section* was introduced in PROGRES 11.0 and is used for redefining existing methods or implementing abstract ones. The redefinition section of the class *ActPerfChecker* consists of two methods: *validate*, which implements the *validate* method of the *Checker* class and *repairAction*, which redefines the method *repairAction* of the *Checker* class. The transaction *checkNextEdge* (presented above) and *eliminateNextEdgeError* (not shown in the paper due to lack of space) are invoked respectively in the *validate* and *repairAction* methods.

```
methods redef
  redef transformation validate =
    self.checkNextEdge
  end;
  redef transformation repairAction =
    self.eliminateNextEdgeError
  end;
end;
```

Fig. 6. The PROGRES class ActPerfChecker - method redefinition section

4 New GraCAD Based on PROGRES UPGRADE

In GraCAD, the prototype for our method, the graph editor for specifying functional requirements for the building to be designed (Fig. 1) is created with the usage of the tool *UPGRADE* (***U****niversal* ***P****latform for* ***GRA****ph-Based* ***DE****velopment*) [2] developed at the RWTH Aachen. UPGRADE is a Java-based framework for developing visual applications which use PROGRES graphs and graph transformations as their internal data model. ArchiCAD 8.0, a commercial system for architects, serves as a CAD basis of our prototype. The graph editor is integrated with an ArchiCAD add-on developed with ArchiCAD General Development Kit 4.3. The editor communicates with this add-on using sockets. In the next couple of months in cooperation with architects we are going to find useful checkers for use case diagrams and implement them in the GraCAD prototype, too.

The right part of Fig. 2 (in section 2) shows a building designed in the ArchiCAD environment. The rooms of this building are marked with *zone* elements. The left part of the picture shows the *graph of zones* for this building. Relations between zones such as accessibility and adjacency are computed within the add-on with the use of ArchiCAD Development Kit mechanisms giving access to the geometrical properties of ArchiCAD elements. Then, appropriate PROGRES transactions are invoked which create the graph of ArchiCAD zones. Having the graph of zones we can check whether this graph matches the room graph specified in the editor of functional requirements. In

the near future we are going to specify the procedure for computing relations between zones with the usage of PROGRES and evaluate if the PROGRES system could be also useful in this field.

5 Related Work

Pioneered by N. Chomsky [6], the linguistic (grammar-based) approach to world modelling has been applied in many areas. The core idea in this methodology is to treat certain primitives as letters of an alphabet and to interpret more complex objects and assemblies as words or sentences of a language based upon the alphabet. Rules governing the generation of words and sentences define the grammar of the considered language. In terms of words, modelling such a grammar generates a class of objects that are considered to be plausible. Thus, grammars provide a very natural knowledge representation formalism for computer-based tools that should aid design.

Since G. Stiny [22] developed shape grammars, many researchers have shown how such grammars allow the architect to capture essential features of a certain style of buildings. However, the primitives of shape grammars are purely geometrical, which restrict their descriptive power. Substantial progress was achieved after *graph grammars* were introduced and developed (cf. e.g. [19]). Graphs are capable to encode much more information than linear strings or shapes. Hence, their applicability for CAD-systems was immediately appreciated [10].

A special form of graph-based representation used for design purposes has already been developed by E. Grabska [11] in 1994. Later on, Grabska's model served as the basic knowledge representation scheme in research reported at conferences in Stanford [12], Ascona [4], and Wierzba [3]. It turned out that by introducing an additional kind of *functionality graphs* into Grabska's model, conceptual solutions for the designed object can be conveniently reasoned about. The additional functionality analysis of houses, as the starting point for the conceptual design, has been proposed by several researchers (compare, e.g. [5], [7]). Such a methodology allows the designer to detach himself from details and to consider more clearly the functionality of the designed object incorporating the constraints and requirements to be met, and the possible ways of selecting optimum alternatives.

In the initial phase of the Polish-German project we have considered the use of the FUJABA [9] and AGG [8] graph transformation systems. Finally, in our research project we have decided to use PROGRES because of the language constructs such as *derived attributes, constraints and repair actions, restrictions, and path expressions*, which are available in PROGRES, but not

in FUJABA and AGG, and are interesting for specifying graph checkers. PROGRES also provides a backtracking mechanism (not available in the other two tool mentioned above) which was used in the project for floor layout generation [13].

The structure of all, or almost all, buildings is hierarchical, therefore, it seems that for representing buildings, the graph transformation system using hierarchical graphs would be useful. However none of the tools mentioned above supports graph transformations on hierarchical graphs. In the past the graph model of AGG was the graph structure of labelled and attributed hierarchical graphs, but unfortunately this model was changed to directed graphs.

In the research of the RWTH AAchen group concerning the usage of the graph rewriting system PROGRES in the area of conceptual design of buildings ([14], [15]), the elicitation phase is skipped and the consistency of a designed building is verified based on the parameterizable graph knowledge specified by a knowledge engineer. In our case we check whether the object to be designed fulfills the graph of requirements specified by a designer in the elicitation phase. The consistency checks are not implemented in ArchiCAD (as in [14], [15]), but are a part of the UPGRADE prototype (specified in PROGRES). The main focus of our work is the transition from the knowledge about the intended use of a building (use cases, activity diagrams, room graphs) to the conceptual design (with the use of PROGRES). In [14] and [15] the main focus is put on the specification of parameterizable graph knowledge for buildings. Therefore, at first glance those two approaches seem to complement each other and it appears that they could be combined, but the differences on a technical level makes such a combination rather difficult (fixed domain knowledge in our PROGRES specification versus generic parameterized specification).

6 Summary

New PROGRES constructs, i.e. packages and methods introduce modularization, extensibility, and reusability of PROGRES graph specifications. The specifications created with the usage of the mechanisms listed above are clearer and simpler. The constructions like *(1)* constructors *(2)* method overloading *(3)* function methods are currently not available. The reasons for not introducing 1-3 into PROGRES are discussed in detail in [16], but from the PROGRES user's point of view, introducing all four constructs would simplify creating specifications in many cases. In comparison to the old tk/tcl based PROGRES graph browser, which we used earlier, PROGRES UPGRADE Framework is much more flexible and allows the creation of complex applications based

on a PROGRES specification. But unfortunately the current version of UPGRADE does not support the recently introduced features of PROGRES, i.e. it is not possible to call a transaction method for a given node displayed in the application built with UPGRADE. Adding such functionality would simplify creating UPGRADE applications considerably. The essential advantage of UPGRADE is that the prototype of visual application generated by this framework for a given PROGRES specification is Java-based. Thus, a user familiar with the Java language can modify the prototype easily.

To summarize, we have presented how the recently introduced extensions of the graph transformation system PROGRES, i.e packages and methods, can be utilized for the specification of constraint checking tools, in particular the conceptual design tool GraCAD. This style of specification, combining object orientation and graph transformations, seems to be very intuitive and useful for specifying graph checkers. The presented modelling method can be adapted by other tools combining the above mentioned paradigms like FUJABA or AGG too.

References

[1] ArchiCAD 8.0 Reference guide, Graphisoft, Budapest, 2002

[2] Böhlen, B., Jäger, D., Schleicher, A., Westfechtel B.: UPGRADE: A Framework for Building Graph-Based Interactive Tools, Proceedings International Workshop on Graph-Based Tools (GraBaTs 2002), Barcelona, Spain, Electronic Notes in Theoretical Computer Science, vol. 72, no. 2 (2002)

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

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

[5] Borkowski, A., Grabska, E., Hliniak, G.: Function-structure computer-aided design model, Machine GRAPHICS & VISION, 9, Warszawa (1999), 367–383

[6] Chomsky, N.: Aspects of Theory of Syntax, MIT Press, Cambridge (1965)

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

[8] Ermel, C., Rudolf, M., Taentzer, G. The AGG Approach: Language and Enviroment, Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages, and Tools, chapter 14. World Scientific, Singapore (1999), 551 - 603

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

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

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

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

[13] Grabska, E., Palacz, W.: Floor layout design with the use of graph rewriting system Progres, In: M. Schnellenbach-Held, H. Denk (Eds.), Proc. 9th Int. Workshop on Intelligent Computing in Engineering, 180, VDI Verlag, Düsseldorf (2002), 149–157

[14] Kraft B., Meyer O., Nagl M.: Graph technology support for conceptual design in Civil Engineering, In: M. Schnellenbach-Held, H. Denk (Eds.), Proc. 9th Int. Workshop on Intelligent Computing in Engineering, 180, VDI Verlag, Düsseldorf (2002), 1–35

[15] Kraft, B., Nagl M.: Parameterizable Specification of Conceptual Design Tools in Civil Engineering, in [18], 90-103

[16] Münch, M.: Generic Modelling with Graph Rewriting Systems, Ph. D. thesis, RWTH Aachen, Aachen (2002)

[17] Neufert, E.: Bauentwurfslehre, Vieweg & Sohn, Braunschweig-Wiesbaden (1992)

[18] Pfaltz, J. L., Nagl, M., Böhlen, B. (Eds.): Applications of Graph Transformation with Industrial Relevance Proc. 2nd Intl. Workshop AGTIVE'03, Charlottesville, USA, 2003, LNCS 3062, Heidelberg: Springer-Verlag (2004)

[19] Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, World Science, Singapore (1997)

[20] Schürr, A., Winter, A. J.: UML Packages for Programmed Graph Rewriting Systems, in: Proc. TAGT'98 - Theory and Application of Graph Transformations, Paderborn, Germany, Nov. 1998, LNCS, Berlin: Springer-Verlag (2000), 396-409

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

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

[23] Szuba, J., Ozimek, A., Schürr, A.: On Graphs in Conceptual Engineering Design, in [18], , 75-89

[24] Szuba, J., Schürr, A., Borkowski, A.: GraCAD - Graph-Based Tool for Conceptual Design, In: A. Corradini, H. Ehrig, H.-J. Kreowski, G. Rozenberg eds., First International Conference on Graph Transformation (ICGT 2002), LNCS 2505, Springer-Verlag, Berlin (2002), 363-377