

Parameterized Specification of Conceptual Design Tools in Civil Engineering

Bodo Kraft, Manfred Nagl

Department of Computer Science III, Aachen University of Technology,
Ahornstrasse 55, 52074 Aachen, Germany
(kraft | nagl)@i3.informatik.rwth-aachen.de

Abstract. In this paper we discuss how tools for conceptual design in civil engineering can be developed using graph transformation specifications. These tools consist of three parts: (a) for elaborating specific conceptual knowledge (knowledge engineer), (b) for working out conceptual design results (architect), and (c) automatic consistency analyses which guarantee that design results are consistent with the underlying specific conceptual knowledge. For the realization of such tools we use a machinery based on graph transformations.

In a traditional PROGRES tool specification the conceptual knowledge for a class of buildings is hard-wired within the specification. This is not appropriate for the experimentation platform approach we present in this paper, as objects and relations for conceptual knowledge are due to many changes, implied by evaluation of their use and corresponding improvements.

Therefore, we introduce a parametric specification method with the following characteristics: (1) The underlying specific knowledge for a class of buildings is not fixed. Instead, it is built up as a data base by using the knowledge tools. (2) The specification for the architect tools also does not incorporate specific conceptual knowledge. (3) An incremental checker guarantees whether a design result is consistent with the current state of the underlying conceptual knowledge (data base).

1 Introduction

In our group, various tools for supporting development processes have been built in the past, for software engineering [1], mechanical engineering, chemical engineering, process control [2], telecommunication systems [3], and authoring support [4], some of them are presented at this workshop. This paper reports about a rather new application domain, namely *civil engineering*.

For all tools mentioned above, we use a *graph-based tool construction* procedure: internal data structures of tools are modeled as graphs, changes due to command invocations are specified by graph rewriting systems. Then, there are two different branches for constructing tools, a research-oriented and an industry-oriented one. In this paper we restrict ourselves to the *research-oriented branch*. There, we derive tools automatically from specifications, using the PROGRES system [5] for specification development, a code generator for producing code out of the specification, and the UPGRADE visual framework environment [6] into which the code is embedded. The resulting tools are

efficient demonstrators for proof of concept purposes. These tools are based on our academic development infrastructure, having been developed in the last 15 years.

Conceptual Design in civil engineering means that design results are elaborated on a coarse and abstract level without regarding details which are later included in constructive design (in other disciplines called detail engineering) [7]. The main goal of conceptual design is to take the various *levels of semantics* for a design problem into consideration (cf Fig. 1): (a) domain specific knowledge, as standards, economy rules, security constrains, or common and accepted design rules, (b) experience knowledge in form of best practice or of using previous design results and, finally, (c) specific user behavior knowledge or wishes, where users are customers or architects, respectively.

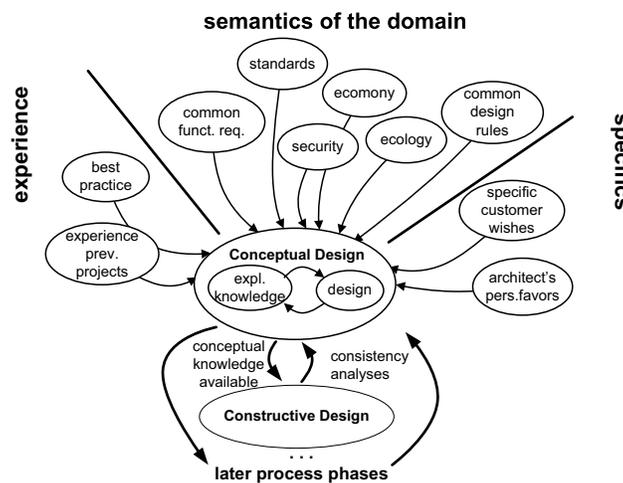


Fig. 1. Different areas of conceptual design

The *essentials* of our conceptual design approach are that (i) explicit knowledge can be formulated, enhanced, or used, (ii) change support is specifically supported, where changes can happen on the level of knowledge as well as for design results, (iii) a lot of consistency checks are included in order to report errors as soon as possible, and (iv) a smooth connection to constructive design is aimed at. The approach specifically pays off, if (v) specific classes of buildings are regarded and, within a class, different designs for buildings and different variants thereof.

We realize a graph-based demonstrator by which a senior architect (knowledge engineer) can specify *knowledge* by tools. The knowledge is specific for a class of buildings. For the usual architect, there are further tools for developing conceptual designs. These *designs* are immediately *checked* against the underlying specific knowledge. For the realization of these tools, we use the enhanced machinery already sketched above. We call this demonstrator the conceptual knowledge experimentation platform as it allows to experiment with concepts without being forced to change the realization of tools.

In this paper we take a certain class of buildings as an *example* namely one-floor medium-size *office buildings*. The example is simplified with respect to breadth and depth. The paper also gives no details of the implementation of tools, only the graph transformation specifications are presented here. Tool functionalities and user interface style of the experimentation platform are given in a separate demo description [8].

The paper goes as follows: In section 2 we give a specification of architect tools in a traditional form, where the building type specific knowledge is fixed within the specification. This motivates the different specification method presented in this paper. In section 3 we discuss the specification for the knowledge engineer tools. Furthermore, we give an example of a host graph which can be produced by interactively using these tools. This graph, called domain model graph describes the characteristics of a class of buildings (here office-buildings). Section 4 gives a specification of the architect's tools by which conceptual designs for buildings can be elaborated. In section 5 we discuss the specification for analyses. Section 6 emphasizes the difference of the two specification methods, the traditional and the parameterized one, summarizes the main ideas of this paper, and discusses related literature.

2 A Traditional Tool Specification

There are many projects in the group using graph technology. The specific knowledge of the appropriate domain usually is hard-wired in the schema and the transaction part of a PROGRES tool specification. In this paper, we apply a *different specification method*.

The *reason* is, that the knowledge engineer will not be able to learn the PROGRES language and to use the realization machinery, adequate for a tool builder. Furthermore, the knowledge should be easily modifiable, as we are experimenting to find suitable object and relation types, restrictions, and rules for conceptual design in civil engineering.

To illustrate the difference between a traditional specification and the parameterized one described here, we briefly introduce an *example specification* which shows how tools for architectural design of an office building would be described in the *traditional* way.

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 to each other by Access and Contains edges. The node class is specialized into five different node types representing different room types we want to model. Therefore, the relations can connect rooms of all specific types. The node class ROOM evidently expresses the similarities of different room node types.

The transaction part determines how different graphs of that graph class are built. Fig. 3 shows a sample *production* for our example. The graph pattern to be searched requires an Outside node to exist and no Entrance_Hall node to be already present (negative application condition). If this pattern is found when applying the production, a new Entrance_Hall node is created and connected with the outside node by an Access edge. So, the application of the production guarantees that the Entrance_Hall is always directly accessible from outside.

Thus, each graph of our example specification models the structure of an office building floor plan. A ROOM node without an Access relation stands for an inacces-

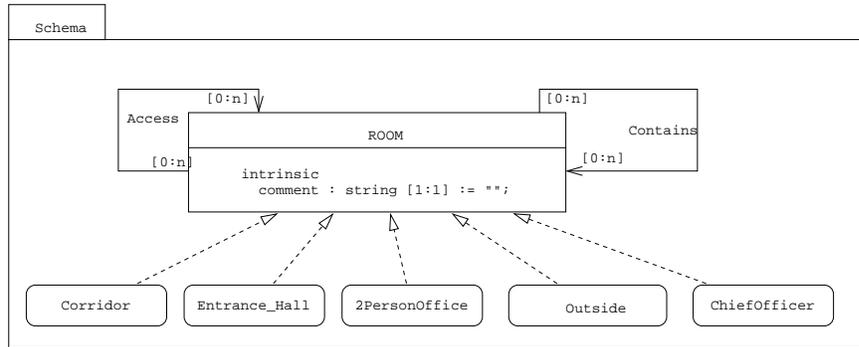


Fig. 2. Schema of an office building

sible room. The *test* shown in Fig. 4 finds such rooms. It searches the graph for ROOM nodes which are not connected with the outside node by a path containing Access or Contains relations. The result is a possibly empty node set. In this way we formally define the meaning of inaccessibility.

We see that the *knowledge* about the building type "office building" is *fixed* within the specification. There are room types Entrance_Hall or 2PersonOffice room defined as node types. Evidently, it is not possible to create new room types, like coffee kitchen with certain accessibility constrains, without *changing* the PROGRES *specification* (schema, transactions, tests). Using PROGRES this way means that the knowledge engineer and the specifier (and later on the visual tool builder) are the same person.

Our request is to keep these jobs separate. The specifier develops a general specification, which does not contain specific application knowledge. This *knowledge* is put in and modified by the knowledge engineer, as a database, here called *domain model*

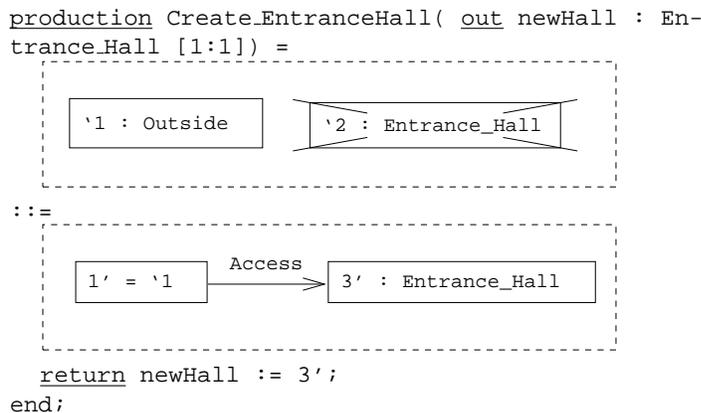
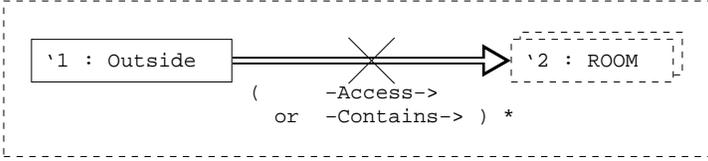


Fig. 3. Example of a graph production, inserting a node and an edge

```

test InaccessibleRooms( out inaccessibleRooms : ROOM
[0:n]) =

```



```

return inaccessibleRooms := '2;
end;

```

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

graph where he is using tools derived from the general specification. In the same way, there is an unspecific specification for the architect tools, from which general tools are derived. The architect tools now use the knowledge domain model graph interactively elaborated by the knowledge engineer. Thereby, the *design* results are *incrementally* checked against the underlying specific knowledge.

It is obvious, that this approach has severe *implications* on how a specification is written, where the domain knowledge is to be found, and where it is used. The different approaches to fix domain knowledge in the specification or to elaborate it in a host graph are not PROGRES specific, they are different ways to specify.

3 Specification of the Knowledge Engineer Tools

In this section we describe the specification for the knowledge engineer tools. Using the corresponding tools, *specific domain knowledge* for a class of buildings is explicitly worked out (domain model graph). This knowledge is used to *restrict* the architecture tools to be explained in the next section.

The upper box of Fig. 5 depicts the PROGRES *schema* part of the knowledge engineer specification. This schema is still hard-wired. It, however, contains only *general* determinations about conceptual design in civil engineering. Therefore, it is not specific for a certain type of building.

The *node class* `m_Element` (`m_` stands for *model*) serves as root of the class hierarchy, three node classes inherit from it. The class `m_AreaType` describes "areas" in conceptual design. Usually, an area is a room. It may, however, be a part of a room (a big office may be composed of personal office areas) or a grouping of rooms (a chief officer area may contain a secretariat, a personal office for the chief officer, and a meeting room).

From the class `m_AreaType` two *node types* are defined; `m_AtomicAreaType` represents an area not further decomposed, `m_ComplexAreaType` a complex area composed of several areas. In the same way, classes `m_Obligatory` and `m_Forbidden` describe obligatory and forbidden *relations*. As we model knowledge about a building type, optional relations are not modeled explicitly. Everything what is not forbidden or obligatory is implicitly optional. The reader may note that attributed relations are rep-

represented in PROGRES as nodes with adjacent edges. Finally, *attributes* may appear as constituents of areas and relations. So, we have again node classes and types to represent the attributes, here only for integer and Boolean values. Note again that attributes have to be defined as nodes, as they are defined by the knowledge engineer.

Fig. 5 in the lower box shows some nodes which stand for kinds of concepts to be used for our office building example. We call these kinds *models*. These models appear in the host graph, interactively produced by the knowledge engineer by using the tools the specification of which we regard. The 2PersonOfficeModel node represents a corresponding room kind, the AccessRelation node an accessibility relation between rooms, the ElectricityAttribute an attribute node needed to describe a property of an office room. These nodes are schematic information (information on type level) for the class of buildings to be described. As, however, this type info is not static but interactively elaborated, we call it model.

An attribute, such as for electric access, may appear in several room models. So, it has to be defined before being used several times. In the same way, accessibility occurs between different room models. Furthermore, it is up to the knowledge engineer, which attribute and relation concepts he is going to introduce. By the way, these definitions may be useful for several types of buildings. Therefore, there is a basic model definition layer in the middle of Fig. 5.

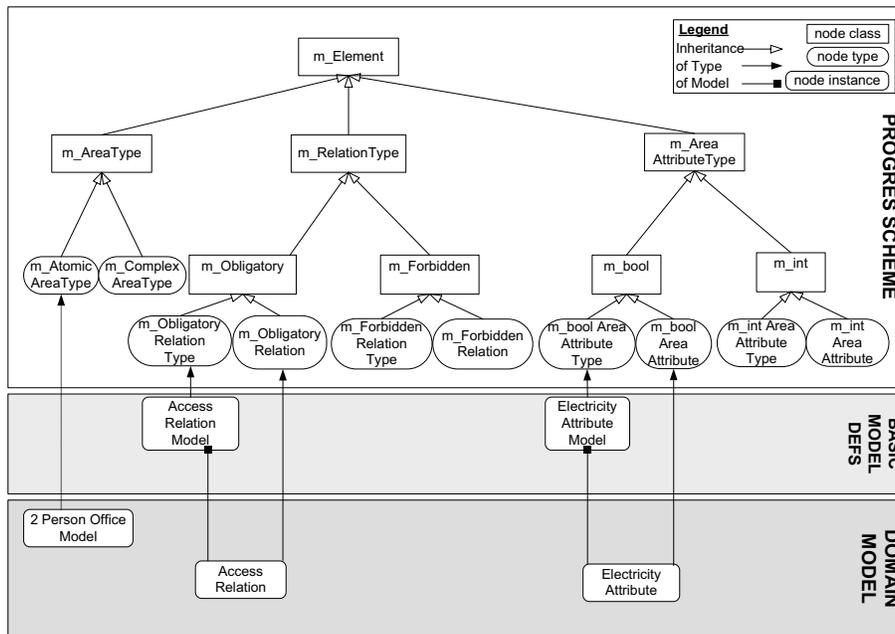


Fig. 5. Schema of knowledge engineering tool and specific model

Summing up, Fig. 5 introduces a 3 level approach for introducing knowledge. The PROGRES schema types are statically defined. They represent hard-wired foundational concepts of civil engineering. The other levels depend on the knowledge tool user. Thereby, the middle layer defines *basics* to be used in the specific knowledge which is dependent on the type of building. So, the static layer on top defines invariant or multiply usable knowledge, whereas the middle layer and the bottom layer are specific for a type of building. The host graph built up by knowledge engineer tools contains information belonging to the middle and the bottom layer.

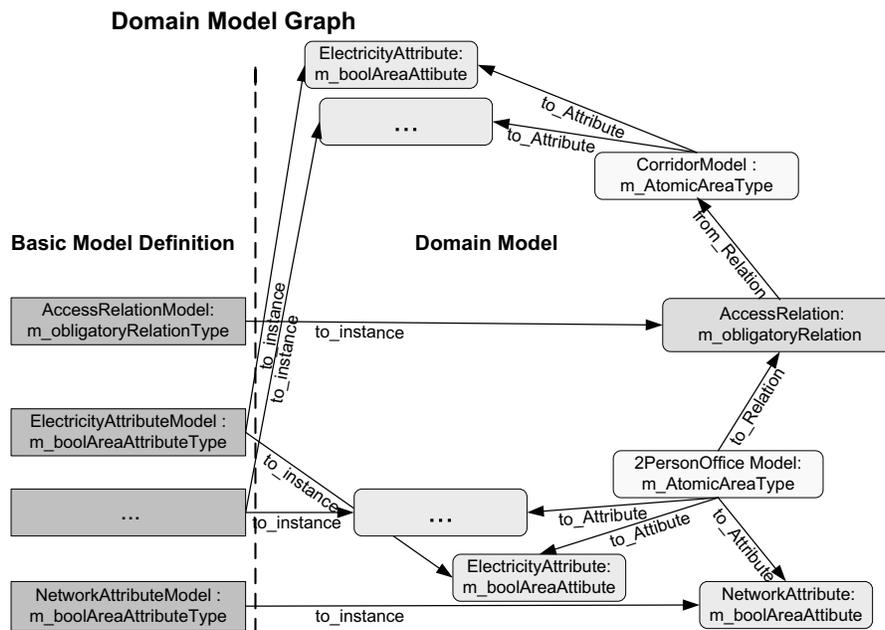


Fig. 6. Cutout of a Domain Model Graph (specific for a type of buildings)

Fig. 6 shows a cutout of this graph structure the knowledge engineer develops, which we call *domain model graph*. On the left side, basic attribute and relation models are depicted. They belong to the level 2 of Fig. 5. On the right side their use in a specific domain model is shown. This right side shows the area models 2PersonOfficeModel and CorridorModel. The 2PersonOfficeModel has two attributes to demand network sockets and electricity to be available. Between the two area models, an access relation is established, to demand an access from all 2 person offices to the corridor, in the graph realized through an edge-node-edge construct.

In Fig. 5 we have introduced a three level "type" system. Any lower level is an instance of an upper level. A node, however, can be an *instance* of the *static* type and a *dynamically* introduced basic *type* as well. We can see that the electricity attribute is

an instance of the static type `m_boolAreaAttribute` and of the dynamic basic type `ElectricityAttributeModel`. This is realized by giving the electricity attribute node a string attribute denoting the dynamic basic type and an edge `to_instance` from the basic type to the attribute node. Tests and transactions guarantee the consistency between these static or dynamic types of instances.

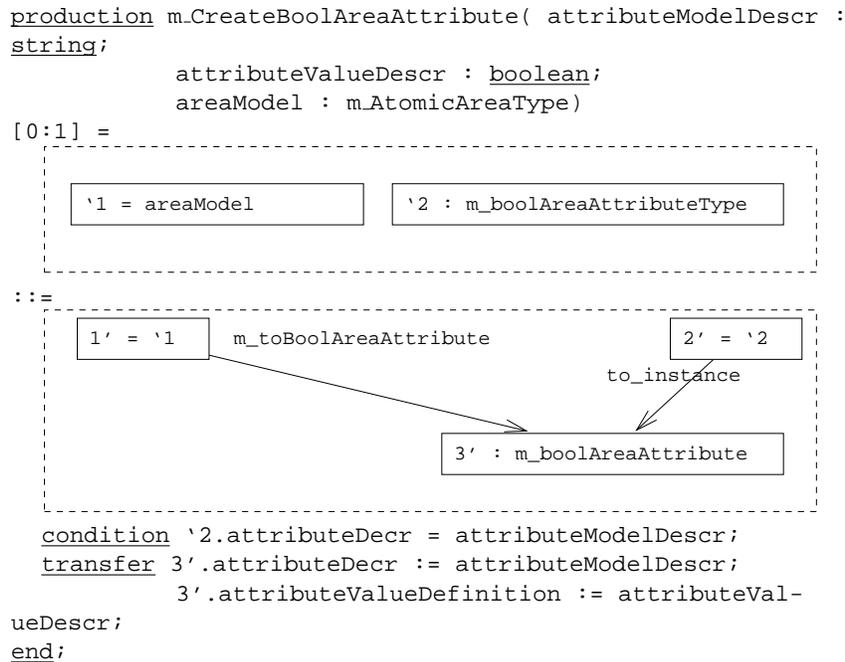


Fig. 7. Creating an instance of an attribute model

Fig. 7 shows a production to create an attribute assigned e.g. to a `2PersonOfficeModel`. Please note that the model is represented by a node with the denotation `areaModel` of the static type `m_AtomicAreaType` which has a `PROGRES` node attribute storing the dynamic type `2PersonOfficeModel`. Input parameters are the attribute model description as a string, an attribute value, and the model node representing the 2 person room concept. Node `'2` on the left side represents an attribute model node. By the condition clause we ensure that it corresponds to the input parameter `attributeModelDescr`. Only if an attribute model (node `'2`) with this description exists, a new attribute (node `3'`) is created and linked to the `2PersonOfficeModel` (node `1'`) and to the attribute model (node `2'`). The model description is stored in a string attribute of node `3'`, just as the attribute value. The inverse operation, to delete an attribute is trivial. Before deleting an attribute model all corresponding instances have to be deleted. This is done by a transaction executing several productions in a specific order.

Interactive development by the knowledge engineer means that transactions modifying the domain model graph are now invoked from outside. Then, this domain model graph is built up containing two levels as shown in Fig. 6. Thereby, the corresponding `to_instances`, `to_attribute`, `toRelation`, and `fromRelation` edges are inserted. Any new concept is represented by a node of a static type (to be handled within the PROGRES system), of a dynamic type, with bordering nodes for the corresponding attributes which belong to predefined attributes of the basic model definition layer.

4 Specification for Architect Tools

Whereas the domain model graph is used to store conceptual knowledge, the *design graph* provides a data structure to represent the conceptual design of a building. The specification of the designer tools directly uses the runtime-dependent basic domain knowledge (layer 2 of Fig. 5). So, the consistency of a design graph with this basic knowledge can be obeyed. The consistency of the design graph with the building type specific knowledge of layer 3 is guaranteed by other analyses. Both analyses are described in the next section.

The design graph allows to specify the *structure* and the *requirements* of a building in an early design phase, above called conceptual design. To design a building without any layout and material aspects allows the architect to concentrate on the usage of this building on a high abstraction level. During the constructive design, this design can be matched with an actual floor plan to discover design errors. This is not further addressed in this paper.

The design graph again is the result of the execution of a PROGRES specification, where transactions are interactively chosen. The 3 level "type" system, which is similar to that of Fig. 5, is shown in Fig. 8. The essential difference is that we now model

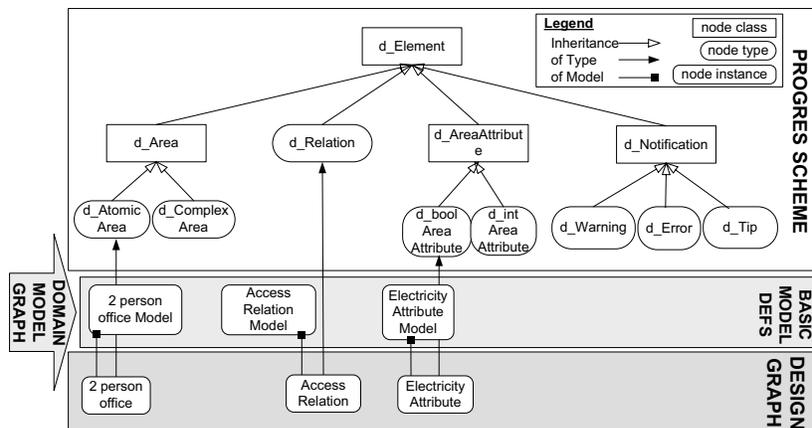


Fig. 8. Scheme of the design graph

concrete objects, relations, both with corresponding attributes and not knowledge describing how such a design situation has to look like. This is denoted by the prefix *d_*, which stands for classes and types for design.

Another *difference* is the *d_Notification* node class with three corresponding node types. The nodes of these types are used to represent warnings, errors, and tips to be shown to the architect. Furthermore, there are now concrete relation nodes between design objects and not rules that certain relations have to exist or may not exist. Finally, the design graph nodes now are instances, and not nodes describing types for instances as it was the case on layer 3 of the knowledge engineer tools.

The *instantiation* of attributes, areas, and relations works in the same way as described in Fig. 7 for models. In the design graph we find instances of concepts with a static and dynamic type with bordering instances of attributes and relations both being applied occurrences of the corresponding basic models introduced on layer 2 of Fig. 5. As this basic model layer is again needed on the design graph level we just import it from the domain model graph.

5 Consistency Analyses

In this section we present *two* different *forms* of consistency *analyses*. The first form is part of the domain model graph specification. So, these analyses are executed when the knowledge engineer tool is running, to keep the dynamic type system consistent. Corresponding internal analyses can be found for the design graph, respectively. The second form of analyses shows how the consistency between the domain model graph and the design graph is checked.

Let us start with the first form of *analyses built in the domain model graph specification*. Fig. 9 shows a test being part of the analyses to guarantee the consistency of the dynamic type system. Each basic model has to be unique. So, if the knowledge engineer tries to create a model that already exists, the enclosing transaction should fail. The test *m_AttributeModelExists* gets as input parameter the model description, e.g. *ElectricityAttributeModel*. If the model already exists, then a node of type *m_boolAreaAttributeType* exists, whose attribute *attributeDescr* has the value of the input parameter.

```

test m_AttributeModelExists( modelDescr :
string) [0:1] =
  '1 : m_boolAreaAttributeType
  valid (self.attributeDescr = modelDescr)
end;

```

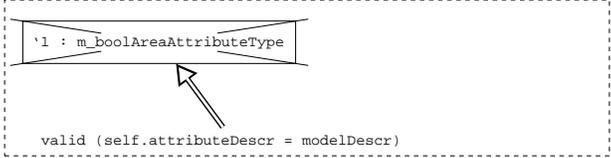


Fig. 9. Test if a model already exists

These analysis transactions *work as usual* in PROGRES specifications. They guarantee that certain structural properties of a graph class (here domain model graph) are fulfilled. In the above example this means that a basic model definition occurs only once. The difference to traditional PROGRES specifications, however, is that the corresponding node *type* is *dynamic*. So we have to check the values of runtime-dependent attributes.

Corresponding internal analyses we also find on *design graph* level, for the consistency between predefined basic knowledge (imported from the domain knowledge graph) and the current form of the design graph. As they work in the same way as the internal analyses of the domain model graph, we skip them.

The second form of analyses check whether there are *violations* of the predefined *specific knowledge* within the *design graph*. For this, we have to find out inconsistencies between the design graph and the domain model part of domain model graph (cf. Fig. 6). The attributes of an area model prescribe the usage of an area in the design graph. In an office block, there should be network sockets in all offices, but not in

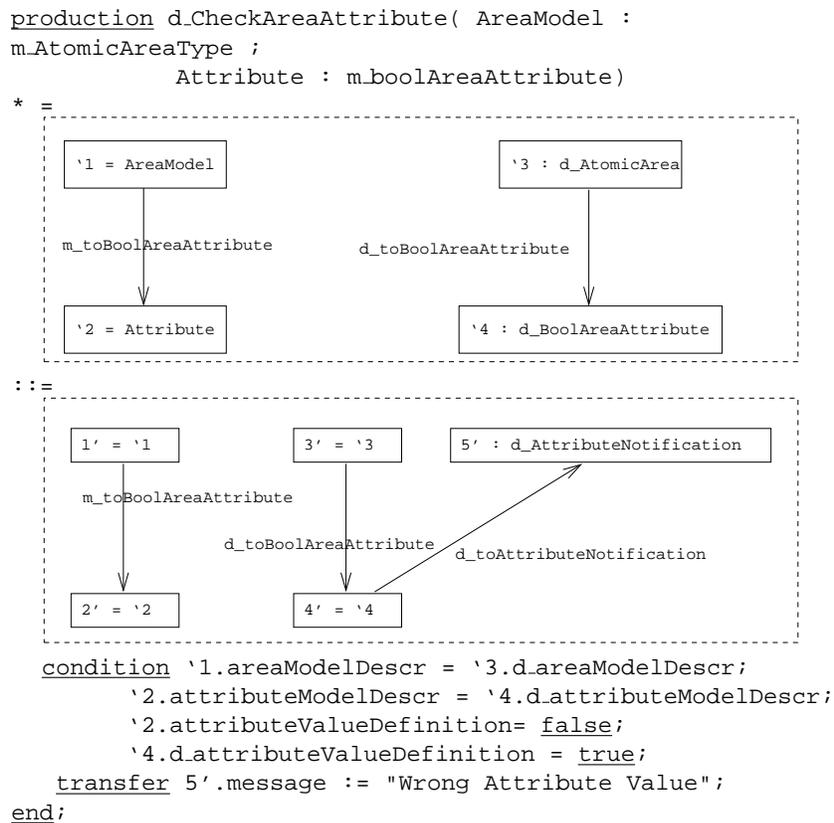


Fig. 10. Analysis to check the consistency of bool attributes

the corridor. This rule is defined in the domain model graph by the Boolean attribute `NetworkAttribute` whose value can be true or false. If the architect constructs a network socket in the corridor, by connecting the area `Corridor` with the attribute `NetworkAttribute`, the design graph is in an inconsistent state.

Tools *immediately report* such inconsistencies. However, we allow the architect to violate rules and do not stop the design process, because we do not want to hinder his creativity.

Fig. 10 shows an example production, which checks whether the value of an attribute, defined in the model graph, corresponds to the attribute value in the design graph. Whereas the nodes `'1` and `'2` describe an area `model` and an attribute defined in the domain model graph, the nodes `'3` and `'4` describe an area and an attribute defined in the design graph. The first two lines of the condition clause ensure that only these nodes of the design graph (node `'3` and `'4`) are found, which correspond to the area `model` (node `'1`) and its attribute (node `'2`). The next two lines of the condition clauses demand the attributes to be `false` in the domain model graph (node `'2`) and to be `true` in the design graph (node `'4`). So, an inconsistency between the *domain model graph* and the *design graph* is found. In this case, on the right side of the production, the new node `5'` is inserted to mark this inconsistency and to store a specific error message.

6 Conclusion and Discussion

6.1 Summary and Discussion

In this paper we introduced a specification method for *tools* in the domain of civil engineering. Different tools provide support for *knowledge engineering* and *conceptual design*, respectively. Analyses within either the knowledge engineer or the architecture tool guarantee internal consistency with the basic knowledge interactively introduced. Furthermore, analyses guarantee the consistency of a design result with the building type specific knowledge. Correspondingly, the *specifications* are split into three parts. The interactively elaborated domain knowledge consists on the one side of a basic part which is useful for several classes of buildings. The *specific* part on the other side represents the knowledge about one class of buildings.

The specification of the knowledge engineering tools allows to introduce *basic model* nodes for attributes and relations. Furthermore, the *specific knowledge* is elaborated by model instance nodes for areas, relations and attributes. The complete information, dependent on the input of the knowledge engineer, is kept in the domain model graph. This information is *used* by the specification for the designer tools, namely by invoking the analyses between designer results and specific domain knowledge.

So, resulting tools are parameterized. In the same way, the (architecture) tool *specification* is *parameterized* in the sense that it depends on specific knowledge to be put in, altered, or exchanged. More specifically, it uses a host graph produced by the knowledge engineer specification. The interactively determined knowledge information can be regarded as dynamic type information.

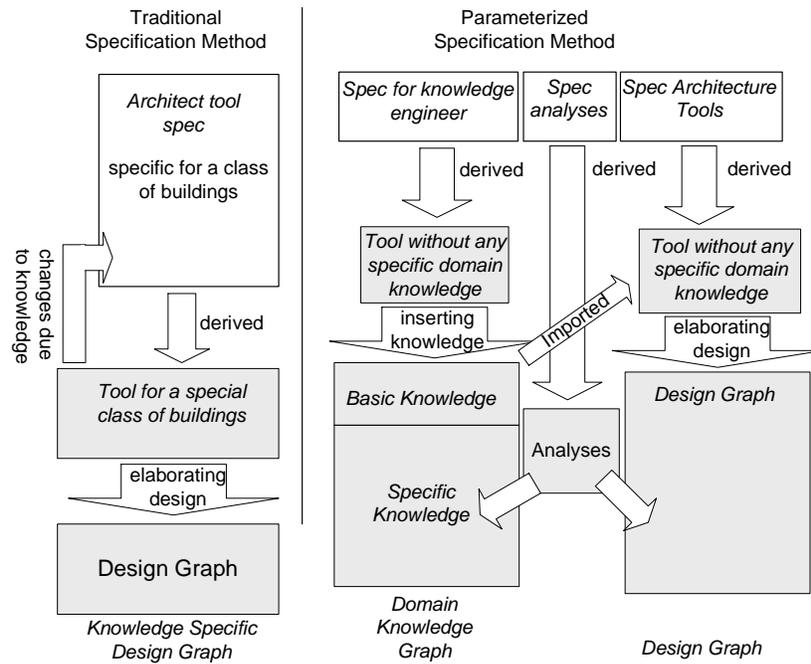


Fig. 11. Traditional and parameterized specification method

Fig. 11 shows both *approaches*, namely the *traditional* and *parameterized* one, to specify tool behavior. In the traditional way (left side) the specific knowledge is contained in the specification of the architecture tool. Whenever the knowledge changes, the specification has to be changed and an automatic tool construction process has to be started. On the right side there is a sketch of the parameterized approach presented in this paper. The knowledge engineer tool has, in its initial form, no specific knowledge. This is interactively elaborated. The resulting host graph (domain model graph) acts as typing information for the architecture tool. The basic knowledge information is imported by the design tool. The specific knowledge information is used for building type-dependent analyses of a concrete design result.

6.2 Related Work in Civil Engineering

Both specification methods have *pros* and *cons*. If the knowledge is fixed, then the traditional way is advantageous. More checks can be carried out at specification elaboration time, which is more efficient. If the underlying knowledge changes, as it is the case with our experimentation platform, the parameterized method is not only better but necessary. Here, changes of the underlying knowledge need no modification of tools. The price is to have more and more complicated checks at tool runtime due to levels

of indirectness which are more costly. Furthermore, the specifications do contain less structural graph information and, therefore, are more difficult to read and write.

Let us now *compare* the results of this paper with other papers in the area of conceptual design in *civil engineering* on one side, and with other graph specification approaches on the other. Let us start with the design literature and concentrate on those which also use graphs. There are several approaches to support architects in design. Christopher Alexander describes a way to define architectural design pattern [9]. Although design patterns are extensively used in computer sciences, in architectural design this approach has never been formalized, implemented and used. In [10] Shape Grammars are introduced to support architectural design, e.g. the design of Queen Ann Houses [11]. The concept of shape grammars is related to graph grammars. However this approach rather supports a generation of building designs than an interactive support while designing, what we propose.

Graph technology has been used by [12], to build a CAD system that supports the design process of a kitchen. In contrast to our approach, the knowledge is hard-wired in the specification. In [13] [14] graph grammars are used to find optimal positions of rooms and to generate an initial floor plan as a suggestion for the architect. Formal concept analysis [15] and conceptual graphs [16] describe a way to store knowledge in a formally defined but human readable form. The TOSCANA systems [17] describes a systems to store building rules.

6.3 Comparison to other GraTra Specification Approaches

Finally, we are going to *relate* our graph specification method to others in the area of *graph technology*. We concentrate on those papers where typical and different tool specification methods are applied. In the AHEAD project [2], a management system for development processes is developed. AHEAD distinguishes between a process meta model, to define the general knowledge hard-wired in the specification, and the process model definition to represent domain specific knowledge, which can be elaborated or changed at runtime. Nevertheless, the tool construction process has to be run again to propagate changes to the AHEAD prototype.

In the ECARES project [3] graph-based tools are developed to support the understanding and restructuring of complex legacy telecommunication systems. The specific domain knowledge consists in this case e.g. of the formal definition of the underlying programming language to be found in a specific specification. As result of a scanning and parsing process a host graph is automatically created representing a system's structure. Changing the specific knowledge, the parser and the specific part of the PROGRES specification have to be adapted and the tool construction process has to restart.

In the CHASID project [4] tools are introduced to support authors writing well-structured texts. Its specification method resembles to the one presented in this paper. The specific domain knowledge is here stored in so called *schemata*, they are again elaborated at runtime. In contrast to our approach, however, the defined schemata are directly used to write texts and not to be checked against a text to uncover structural errors. So, the main advantage of the new specification method of this paper is a gain in flexibility!

References

1. Nagl, M., ed.: Building Tightly Integrated Software Development Environments: The IPSEN Approach. Volume 1170 of Lecture Notes in Computer Science. Springer, Berlin (1996)
2. Jäger, D., Schleicher, A., Westfechtel, B.: AHEAD: A graph-based system for modeling and managing development processes. In Nagl, M., Schürr, A., Münch, M., eds.: AGTIVE'99. Volume 1779 of Lecture Notes in Computer Science., Kerkrade, The Netherlands, Springer, Berlin (2000) 325–339
3. Marburger, A., Herzberg, D.: E-CARES research project: Understanding complex legacy telecommunication systems. In: Proceedings of the 5th European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, IEEE Computer Society Press, Los Alamitos, CA, USA (2001) 139–147
4. Gatzemeier, F.: Patterns, Schemata, and Types — Author support through formalized experience. In Ganter, B., Mineau, G.W., eds.: Proc. International Conference on Conceptual Structures 2000. Volume 1867 of Lecture Notes in Artificial Intelligence., Springer, Berlin (2000) 27–40
5. Schürr, A.: Operationales Spezifizieren mit programmierten Graphersetzungssystemen. PhD thesis, RWTH Aachen, DUV (1991)
6. Böhlen, B., Jäger, D., Schleicher, A., Westfechtel, B.: UPGRADE: A framework for building graph-based interactive tools. In Mens, T., Schürr, A., Taentzer, G., eds.: Electronic Notes in Theoretical Computer Science. Volume 72 of Electronical Notes in Theoretical Computer Science., Barcelona, Spain, Elsevier Science Publishers (2002)
7. Kraft, B., Meyer, O., Nagl, M.: Graph technology support for conceptual design in civil engineering. [18] 1–35
8. Kraft, B.: Conceptual design tools for civil engineering, demo description, this workshop (2003)
9. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S.: A Pattern Language. Oxford University Press, New York, NY, USA (1977)
10. Gips, J., Stiny, G.: Shape grammars and the generative specification of painting and sculpture. In: Proceeding of the IFIP Congressn 71. (1972) 1460–1465
11. Flemming, U.: More than the Sum of Parts: the Grammar of Queen Anne Houses, Environment and Planning B . Planning and Design (1987)
12. Göttler, H., Günther, J., Nieskens, G.: Use of graph grammars to design cad-systems. In: Graph Grammars and their application to Computer Science, LNCS 532, Springer, Berlin (1990) 396–409
13. Borkowski, A., Grabska, E., Szuba, J.: On graph based knowledge representation in design. In Songer, A.D., John, C.M., eds.: Proceedings of the International Workshop on Information Technology in Civil Engineering, Washington (2002)
14. Borkowski, A., Grabska, E., Nikodem, E.: Floor layout design with the use of graph rewriting system progres. [18] 149–157
15. Stumme, R., G. Wille, E.e., eds.: Begriffliche Wissensverarbeitung, Springer, Berlin (2000)
16. Sowa, F.J., ed.: Conceptual Structures. Addison Wesley, Reading, MA, USA (1984)
17. Eschenfelder, D., Stumme, R.: Ein Erkundungssystem zum Baurecht: Methoden und Entwicklung eines TOSCANA Systems. [15] 254–272
18. Schnellenbach-Held, M., Denk, H., eds.: Advances in Intelligent Computing in Engineering, Proceedings of the 9th International EG-ICE Workshop, Darmstadt, Germany (2002)