Using Graph Rewrite Systems for supporting the Software Design Process by finding suitable Software Components

Bettina Sucrow and Torsten Heverhagen

Data Management Systems and Knowledge Representation, Dept. of Mathematics and Computer Science,

University of Essen, Germany

{sucrow | the}@informatik.uni-essen.de

ABSTRACT

Software engineers have to decide permanently during the software design process whether to specify every necessary software portion of the system to be designed entirely themselves or, rather, to search for and possibly substitute a corresponding suitable software portion already existing and functioning well.

Our approach investigates the latter possibility in the following sense: if we presume that software engineers specify their software components in a similar way then it should be possible to compare only abstractly specified components with already concretely specified ones with the goal of substituting the former by the latter, respectively, rather than specifying the whole system in every detail. However, such an approach has only a chance to really work correctly if it is possible to describe a software design process of such a kind by a method offering an intuitive understanding, *a* formal basis *and even, finally, a* practical result. *Our idea is to use graph rewrite systems for this purpose.*

1 Introduction

Software engineers have to decide permanently during the software design process whether to specify every necessary software portion of the system to be designed entirely themselves or, rather, to search for and possibly substitute a corresponding suitable software portion already existing and functioning well. In this contribution we propose to support the software engineer during the software design process by the possibility of search for and substitution of suitable software components wrt the current design. The idea is that the designer specifies some portions of the system to be developed very abstractly with the goal of using these as templates or patterns in order to search for corresponding concretely specified software components to be integrated into the design.

Such a process can only function under the presumption that it can be expressed by a suitable formalism. This means the respective formalism to offer understandability in order to preserve the intuitive idea of the given problem, a mathematical foundation in order to allow correctness and consistency proofs whenever desired or needed, and, finally, the practical side, namely, the possibility to automate the whole process. It will be shown in this paper that graph rewrite systems nicely fulfil all these requirements.

Graph rewrite systems have been used successfully as specification formalism in the software design area in various ways. Engels et. al. (1983) and Engels et. al. (1989) have described a software development environment with graphs as central data structures for the programming language Modula-2. Here, graphs are related directly to the syntax graphs defining the underlying programming language. Goedicke (1993) uses graph grammars for describing module concepts in structured and object-oriented programming languages focusing on the semantics of such concepts. Richter (1995) uses graphs representing object-oriented source code in order to conduct object-oriented design from an abstract level to a detailed level of implementation. Besides the so-called class structure graphs, however, graph rewrite rules specifying reuse guidelines are only mentioned as a topic for future work.

Graph grammar descriptions have also been used in the area of graphical user interface specification. In Arefi et. al. (1991) a customized user interface design environment is generated. First, a conceptual framework for task-oriented user interface specification is specified as a visual language. The specification is then applied to a visual language generator so that a visual syntax-directed editor for the specification language is generated. In this approach the visual language is specified with graph transformation systems. Specification and representation of user interfaces based on end user tasks using attributed graphs and related graph rewriting systems may also be found in Freund et. al. (1992). In order to achieve expressive as well as understandable graphical user interface specifications graph grammars have been used in Goedicke and Sucrow (1996) where graphs describe dialogue states and graph rewrite rules dialogue state transitions. In Nowak et al. (1996) and Sucrow (1996A) this approach has been used in order to formalize a part of the graphical user interface of a complex numerical computation system.

The formal notation of graph grammars is also suitable for achieving a continuous specification process between the requirements and the design stages. Sucrow (1996B) and Sucrow (1998A) show an approach for integrating

software-ergonomic aspects in formal specifications of graphical user interfaces in order to improve the comfort for the user. The idea is to describe a graphical user interface by a still abstract graph grammar in early specification stages and to relate this specification step by step to a concrete graph grammar description specifying additionally specific software-ergonomic features like certain metaphors, etc. This idea of relating abstract graph grammar specifications with more concrete ones could be extended in Sucrow (1997). Here, human-computer interaction is specified by a graph grammar already at very early specification stages, and an approach is presented how this abstract graph grammar can be refined successively to a desired concrete one by applying certain graph rewrite rules. For this purpose of refinement an approach has been presented in Sucrow (1998B) allowing to refine graphs as well as graph rewrite rules specifying a still abstract interactive system to a more concrete one by applying specific graph rewrite rules at a certain meta level.

However, in neither of these approaches graph rewrite systems have been used to support the software design process by searching purposely for already concretely specified software components and substituting them for still abstractly specified ones in a software system to be specified. This goal is realized in our approach: the two steps of applying a graph rewrite rule -- the matching of its left hand side into the graph to be modified and the subsequent substitution of the match by its right hand side -- are used for the search for a suitable component desired wrt a particular software design existing so far and for the substitution of such a component at the correct place in this design respectively.

This contribution is organized as follows. In the next chapter 2 the idea of our approach is presented. Additionally, some necessary preparations are considered wrt such a software design process. In chapter 3 an example specification is introduced for demonstrating our approach of search for and substitution of software components during the software design process. In chapter 4 we present the formal basis, that is, we show how our approach of search for and substitution of software components can be specified formally by graph rewriting according to the algebraic double pushout approach. The practical side then is discussed in chapter 5 where it is demonstrated how correct graph rewrite rules realizing the search and substitution process according to our idea can be generated. Even, an automation wrt to tool support is conceivable. In chapter 6 main challenges resulting for future research are presented and, finally, we come to conclusions.

2   Idea and Preparations

In this paper a new approach is proposed for supporting the software engineer during the software design process. The idea is to allow particular portions of the intended software system to be specified only very abstractly. In order to concretize these parts of the system it shall be possible to search for corresponding, concretely specified and already existing software components. For this search the abstractly specified component will be used as a search pattern. In case of finding a corresponding concretely specified component it shall be possible to substitute it for the original pattern within the system to be designed.

In order to achieve a really helpful support for the software design process our approach shall be applied already in early design stages. Additionally, it is a necessary condition that the software portions to be searched for as well as the corresponding search patterns, the abstractly specified components, have to be comparable up to a certain degree. According to these two requirements we assume the software engineers to specify their intended systems with the aid of an object-oriented design method. Also, the software components to be searched for are assumed to be specified by an object-oriented design method. Correspondingly, the notations will be object-oriented, of course, and of either kind like OMT (cf. Rumbaugh et.al. (1991)), OOSE (cf. Jacobsen et.al. (1992)), OOA/OOD (cf. Booch (1994)), UML (Unified Modeling Language, cf. Booch et.al. (1997A)) as a merging of the former three, and several others.

There are some important problems which will not belong to the focus of this paper. One of those is related to how the various object-oriented design notations can be *unified* in such a manner that software components specified by either of those are still comparable. Similarly, a second one concerns the fact that two software components, a search pattern and a corresponding detailed described component, may have been specified using different vocabularies. In such a case according to our idea the latter shall be substituted for the former one within the system to be specified *despite* this fact. For these and similar problems there exist solution ideas already, which will not be discussed in detail here but belong to ongoing work and are sketched in chapter 6.

In order to introduce our approach in a more detailed way we will demonstrate a small case study using the object-oriented and multiple used notation UML. Suppose, the designer specifies a portion of a bank system where a bank server serves banks possessing accounts which, in turn, can be accessed by customers via automatic teller machines. Then, with the aid of an abstractly specified component describing an automatic teller machine, it is searched for a corresponding concretely specified component. After finding a respective suitable component it is substituted into the system for the abstract one existing so far.

The approach presented here may be used later on in a realistic environment like the internet in the following

manner. Imagine software designers sitting at the screen and specifying their systems. If such a designer specifies a bank system as mentioned above a component like an automatic teller machine may be specified only very abstractly with the goal of using this as a search pattern for a particular internet search machine. Such a machine according to our approach would search for a corresponding concretely specified automatic teller machine by using the given pattern for comparison. After finding a suitable component the machine would suggest it to the designer for substitution, the specification as well as the corresponding source code.

In this paper, however, the focus lies on the theoretical foundations of search for and substitution of desired software components according to the idea presented above. In the next chapters a case study concerning the specification of a bank system is demonstrated. The specification process will be described in an *intuitive* manner by presenting the UML specification process first, in a *formal* manner by realizing this approach with the aid of graph rewriting according to the algebraic double pushout approach and, finally, in a *practical* manner by generating a corresponding graph rewrite rule allowing, finally, the automation of the intended search for and substitution of an automatic teller machine.

## 3   Graph Rewriting as an *intuitive* Basis

The following UML class diagram specifies a part of a bank system. A bank server may serve at least no or at most many banks. The class `BankServer` has three methods `registerBank`, `unregisterBank`, and `readCustomerInfo`, the last of which has three formal parameters `KNo`, `BNo` and `PIN` of type integer. The last method also allows, e.g., a component ATM denoting an automatic teller machine to offer informations about a specific customer, that is her or his account code, code of the bank and PIN number to be tested by the bank server. Vice versa a bank may serve at least no or at most many bank servers. The class `Bank` besides methods has variables `name` and `address` identifying the institute. As can be seen there are also classes specifying accounts and customers having specific relationships to the bank and to each other respectively.
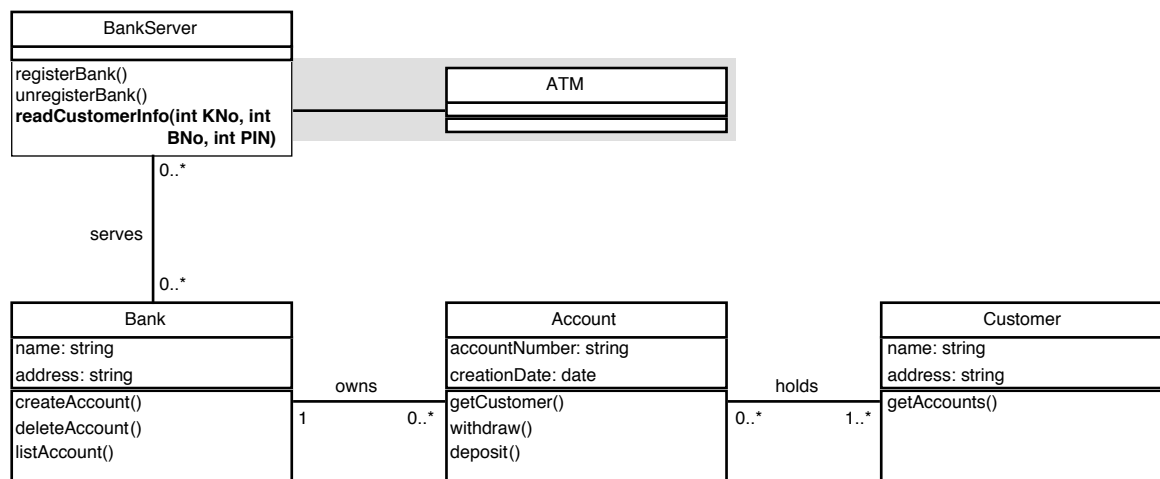

As may be observed there exist parts in the class diagram which are presented differently. The grey shadowed part, an only very abstractly specified component `ATM` together with a still unspecified relationship to the `BankServer`, denotes a part of a search pattern. The designer has specified only very abstractly the desired component realizing an automatic teller machine with the goal of using this subspecification as search pattern in order to find a corresponding concretely specified component. Additionally, the search pattern contains also the method `readCustomerInfo` presented in bold font with its formal parameters `KNo`, `BNo` and `PIN` of type integer specifying the interface required between an automatic teller machine and the remaining system.

Imagine now the designer searches for a concretely specified component realizing an automatic teller machine under the condition that it fulfils the connection to the interface as described above. The respective search pattern would be specified by the parts presented grey shadowed and in bold font above. We expect the designer to possibly find the desired software component somewhere where it may be offered. Such a component could be specified like in the UML class diagram below. It shows the desired component consisting of the grey shadowed part again. Here, the automatic teller machine is denoted `ATMachine` and is extended by two methods. Further, it has associations to other classes specifying the device for reading the cheque card, the safe both of which have inherited from a class specifying an ID device, and the user interface. Thus, the grey shadowed part of the specification represents a concretely specified component. The fact that `ATMachine` is another word compared to `ATM` in the search pattern shall not disturb here. As mentioned above we are currently investigating several approaches allowing to compare semantically equivalent components although they may be possibly expressed using different vocabularies (cf. chapter 6). The same yields for the interface `readcustomerinfo` together with its three formal parameters `kno`, `bno` and `pin` of type integer, which behaves

exactly like the corresponding one specified by the search pattern.
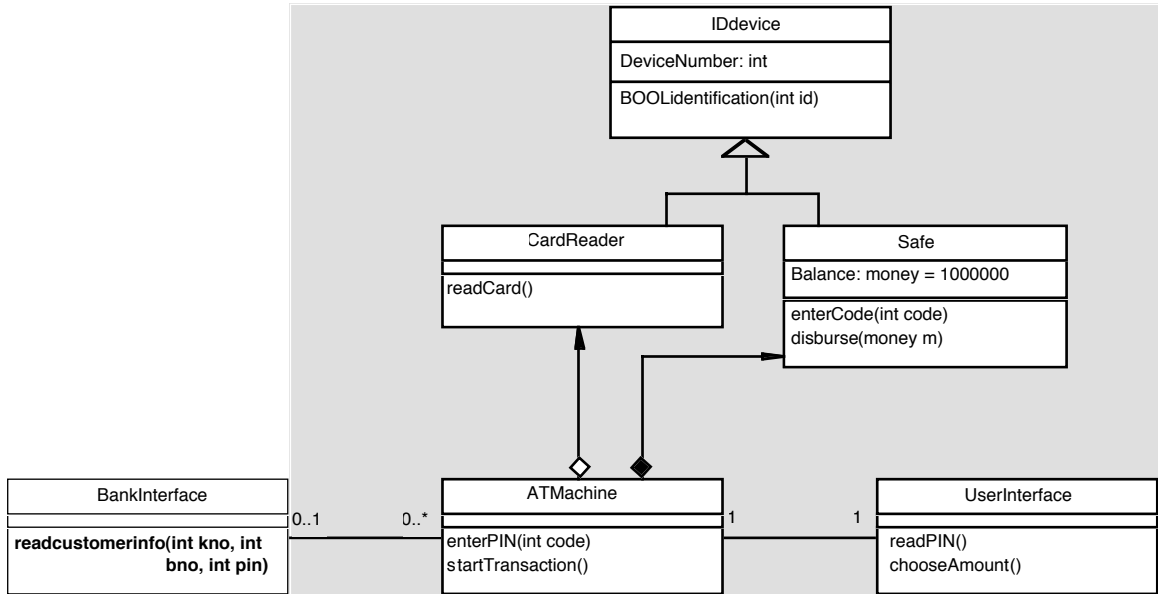
The component in the diagram above is found due to the specification of the search pattern containing both aspects, the abstractly specified component ATM as well as the complete interface `readCustomerInfo`. Suppose now, the concretely specified component is substituted into the currently developing system, then this would result in the following extended specification containing now the found component instead of the class ATM under the condition that the interface `readCustomerInfo` is used by class `ATMachine`.
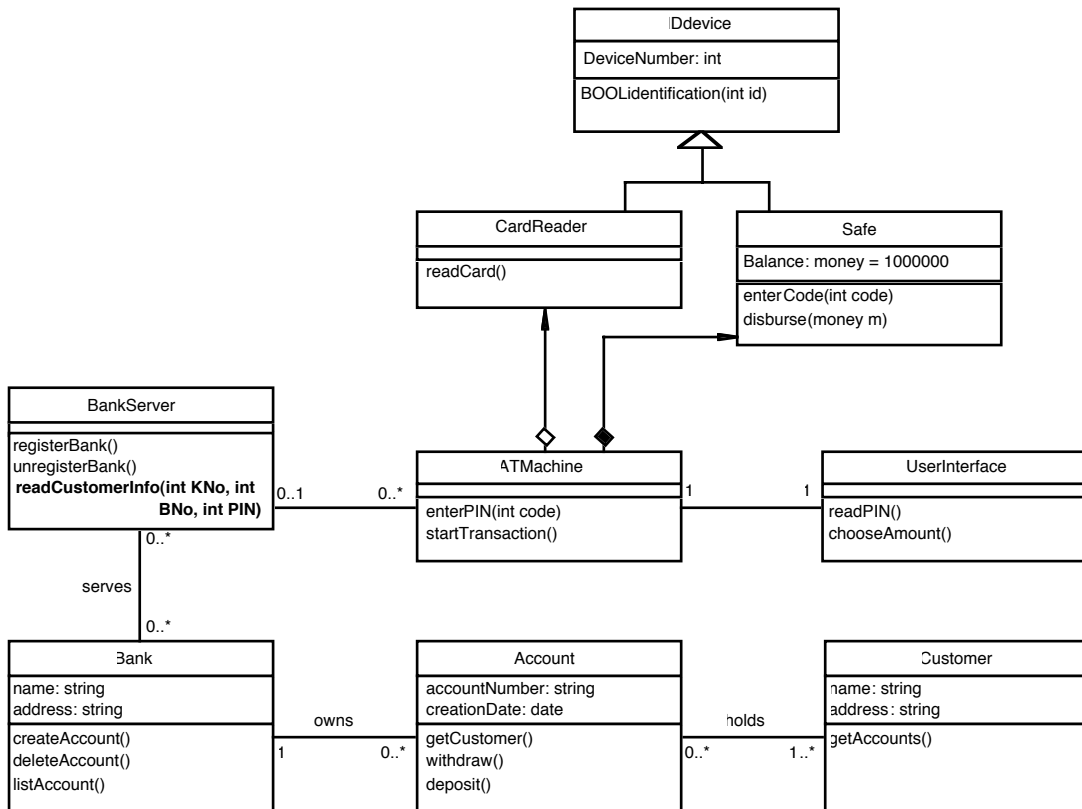
Considering the above described step of software development leads to the following observations:
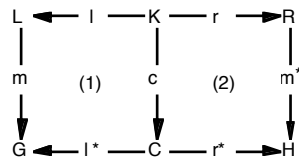
- UML class diagrams are nothing else than graphs.
- Changes of UML class diagrams are nothing else than graph rewriting.
- A search pattern may be characterized by the left hand side L of a graph rewrite rule, and its application for the search process by matching L in a graph to be modified. The right hand side R of this graph rewrite rule may characterize the part that may be substituted for the match.

These observations encouraged us to use graphs and graph rewriting as an intuitive, a formal and, finally, as a practical foundation for searching concretely specified software components with the aid of abstractly specified ones and substituting them for the latter ones into the system to be specified. In the next chapter 4 the

algebraic double pushout approach will be used in order to formalize the process described above.

4   Graph Rewriting as a *formal* Basis

We use graph transformation according to the algebraic double pushout approach (cf. Rozenberg (1997)). Modifications of graphs by a rewrite rule according to this approach are modelled by gluing constructions of graphs that are formally characterized as pushouts in suitable categories having graphs as objects and total graph morphisms as arrows. A graph rewrite rule p is given by a pair of graph homomorphisms from a common interface or gluing graph K (see figure below), and a direct derivation consists of two gluing diagrams of graphs and total graph morphisms, see diagrams (1) and (2). The context graph C is obtained from the given graph G by deleting all elements of G which have a pre-image in L but none in K. This deletion is modelled as

$$
\begin{array}{ccccc}
L & \xleftarrow{\;l\;} & K & \xrightarrow{\;r\;} & R \\
\downarrow m & (1) & \downarrow c & (2) & \downarrow m^* \\
G & \xleftarrow{\;l^*\;} & C & \xrightarrow{\;r^*\;} & H
\end{array}
$$

an inverse gluing operation by diagram (1), while the actual insertion into H of all elements of R which do not have a pre-image in K is modelled by the gluing diagram (2). The match m must satisfy the so-called gluing condition which takes care that the context graph C will have no dangling edges and that every element of G that should be deleted by the application of p has only one pre-image in L. In our approach it is intuitively very helpful as well as formally correct to identify interfaces in the context of search for and substitution of software components as described in the previous chapter 3 as parts of the gluing graph K.

After the decision of using the double pushout approach as formal specification method the next step is to correctly map graphs and graph rewrite rules onto our problem domain. This means to correctly specify UML class diagrams by graphs and changes of UML class diagrams by graph rewrite rules as observed already in chapter 3. Besides our goal of specifying any object-oriented design formally by graphs and graph rewriting as mentioned in chapter 2 already it should be pointed out that many researchers currently are investigating the semantics of UML. One of the most interesting contributions from our point of view is presented by Gogolla and Parisi-Presicce (1998) who show how to transform UML state diagrams into graphs by making explicit the intended semantics of the diagrams. Other approaches are also refered to in this paper. For the purpose of our approach, however, a formalization of UML class (and later also other) diagrams as well as a formalization of changes of those by suitable graphs and corresponding graph rewrite rules, respectively, is needed.

### 4.1   Statics: Graphs specifying UML Class Diagrams

In the following picture an intuitive and understandable description of the class `BankServer` on the left side is given by the graph on the right side. The class is specified by a node of a particular type sketched by a rec-

tangle and identified by the name of the class. The node has a directed edge to a node of another type specifying the method `readCustomerInfo` the formal parameters of which are attached as attributes to it together with their respective types. Similarly, the relationship between the two classes `BankServer` and `Bank` below may be specified by the following graph connecting the two graphs describing the two classes by two
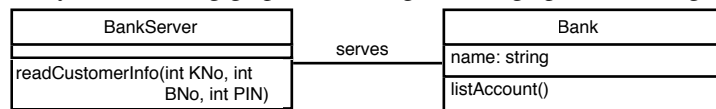

directed edges respectively. The edges are attached by the attribute *serves* specifying the kind of relationship the two classes have to each other.
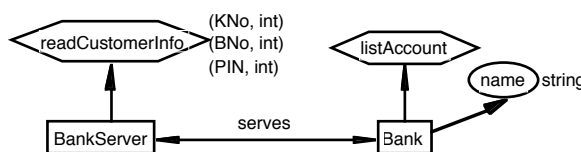

With these intuitive formal specifications of UML class diagrams by graphs it is now possible to specify the dynamics relating to search for and substitution of software components during the software design process by graph rewrite rules.

4.2   Dynamics: Graph Rewrite Rules specifying Changes of UML Class Diagrams

Specifying the system designed so far by a software engineer as described by the first UML class diagram in chapter 3 results in the following graph G where the nodes specifying the classes Bank, Account and Customer, respectively, are not fully specified due to clarity. The two parts represented in bold font and grey shadowed in the UML diagram are easily identifiable in graph G. These two together constitute the search pattern so that they may be described formally as the left hand side L of a particular graph rewrite rule. This is due to the nice possibility of using this graph L for the necessary match in order to find a more concretely specified corresponding component. Further, the part in bold font describing the required interface (cf. chapter 3) may be specified as an anchor, the graph K of this particular graph rewrite rule, which must not be changed by a possible later substitution. Due to clarity, the graphs L and K are presented below additionally.
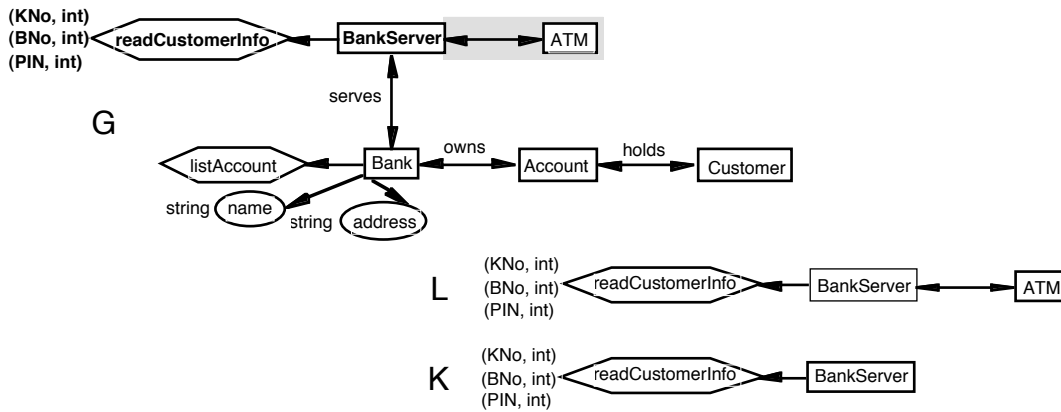
Correspondingly, the found component (cf. the second UML class diagram in chapter 3) looks like the following graph where not every node specifying a class is entirely described again, due to clarity. Here, the interface
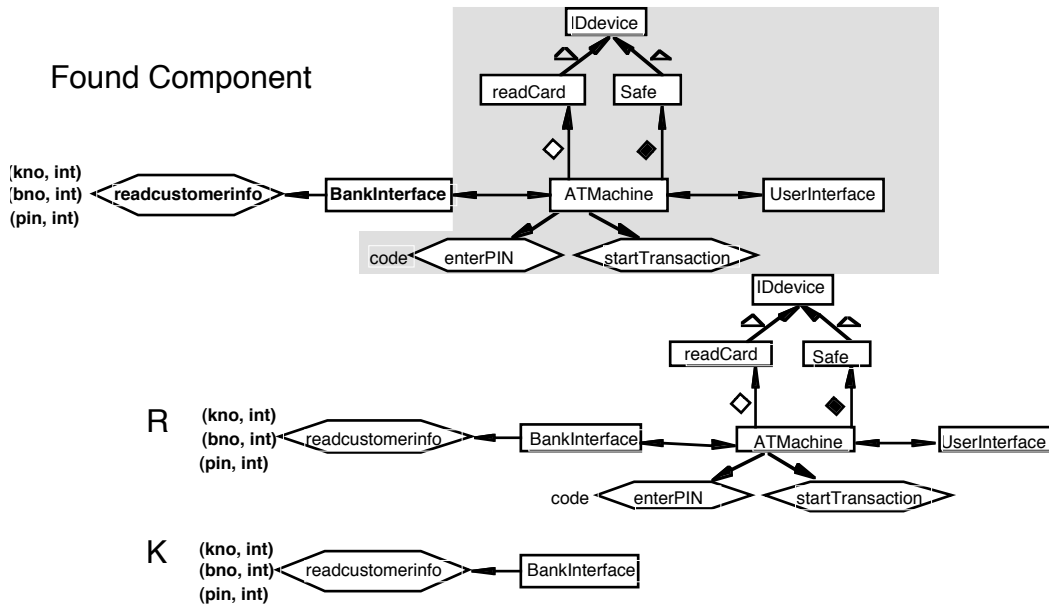

again appears in bold font and is not to be substituted as opposed to the grey shadowed part of the found component. If we think about a possible substitution it is intuitively helpful to identify the found component by the right hand side R of the graph rewrite rule described above where R also integrates the part in bold font. Again this latter part may play the role of an anchor element necessary as requirement for finding the component on the one side, but which must not be substituted on the other side. Again, graphs K and R are sketched additionally above. It should be repeated at this point that different words within two graphs to be compared do not result in a problem due to our investigations wrt unification of words in a certain sense (cf. chapter 6).

A substitution of the found concretely specified component for the abstractly specified one used as search pattern would look like the graph H (cf. the third UML class diagram in chapter 3) below where the original interface has remained as intended.

Obviously, the above described step during the software design process may be specified formally by a graph rewrite rule p consisting of the above introduced three parts L, K and R according to the double pushout approach. However, the rule p is not yet known during the stage of searching. But, as can be observed, every single part of the rule is evolving early enough before its necessary application. Thus, a graph rewrite rule p

realizing the search for and substitution of a software component according to our approach may be generated

H
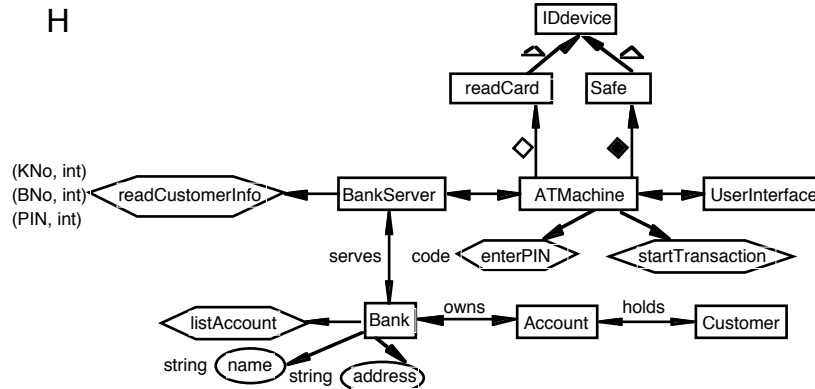
successively so that, finally, this step of the specification process even may be automated (cf. next chapter 5).

5 Graph Rewriting as a *practical* Basis

In chapter 3 we presented an important step of the software design process according to our idea of search for and substitution of software components in an intuitive manner. In the last chapter 4 it has been shown how this step can be formalized. However, this approach is only helpful if it can be applied practically. Thus, it has to be investigated how a graph rewrite rule p realizing the above described steps could be generated during the evolving software design process. According to the intuitive as well as formally correct proceeding described above this could happen in the following way, G being the graph currently existing as a particular design.

Graph Rewrite Rule Generation

- Determine in G graph L specifying the abstractly specified component in comparison of which a corresponding concretely specified one has to be searched for.
- Determine in subgraph L of G graph K that is the interface or anchor elements.
- Search for L in offered components that is perform a graph match with the aid of L.
- Try to find a concretely specified component according to the match by graph L. Identify a possibly found match as graph R to be substituted within graph G for graph L where graph K remains unchanged by definition.

This approach works under the assumption that the found component is offered as a whole and not as subpart of a more complex system. It is assumed that components are offered exactly for the purpose of searching for them in order to substitute them into a software system being currently under design.

With the aid of the above described rule generation it is possible to construct algorithms realizing our software design approach. A sketch of an algorithm is given subsequently in a very coarse manner.

Automation

- Delete within graph G
  - * all connections between graph L and graph K
  - * $L \backslash K$
- Add within graph G
  - * $R \backslash K$
  - * all connections between graph R and graph K

We expect it to be possible of using an algorithm like the one given above to realize the search for and substitution of concretely specified software components for the corresponding abstractly specified ones within a software system to be specified. Of course, some problems still are to be solved as mentioned in chapter 2 and discussed in the next chapter 6.

6 Main Challenges for Future Research

As indicated in chapter 2 already there exist some problems wrt comparison of software components. Due to our goal of specifying search patterns in very early design stages it can not be expected to get exact matches between those and existing component descriptions. Therefore, similarities between specifications have to be investigated. Finding and comparing component specifications, even possibly by automated tool support later on, requires mechanisms for computational detection of similarity. First approaches wrt such mechanisms are

performed in the area of analogical and case-based reasoning (cf. Spanoudakis et.al. (1996), Bergmann and Eisenecker (1995)). The main concept of these approaches is the computation of distance or similarity metrics. In our approach a component is expected to be described not only by its interface but also by its contents. The description contains component artifacts, that is all UML model elements used to describe components. An overall distance metric then is an aggregation of the distance metrics of all component artifacts.

One of the main problems of characterizing similarities is the acquisition of semantical information about software components. Spanoudakis et.al. (1996) claim that similarities between component artifacts can be detected using their classification, generalization and attribution. Generalization and attribution can automatically be extracted from the most object-oriented design models. Classification is expressed through *instance-of* relations. Many object-oriented models support the *instance-of* relation only between simple instances and their classes. What we need is a notion of *meta classes*, that is classes at a certain meta level, in order to classify classes as instances of such meta classes. For the first time UML introduces a *meta model* (cf. Booch (1997B)) providing information by classifying component artifacts according to classes, states, operations etc. Further, classification can be achieved by using UML *stereotypes* which can be seen as user defined extensions to the UML meta model. There already exist predefined stereotypes like *actor*, *interface object*, *entity object* or *control object* the use of which can improve classification dramatically.

A very important attribute of component artifacts is the component's name containing useful semantical information. Unfortunately, it is not unique in that software developers may use different vocabularies as well as synonyms or homonyms (the opposite of synonyms). For this reason we plan to integrate "of-the-shelf" and domain specific lexical databases to improve the usability of names during the search. These databases like WordNet (cf. Miller et.al. (1990)) already contain relations like synonyms, antonyms, hyponyms etc.

The requirements for characterizing similarities between software component specifications are complex wrt our goal of substituting found component specifications into the software specification existing so far, especially, when considering also automated tool support for the future as well. Some future work still has to be done in order to realize practically the ideas presented in this paper in complex and realistic software design processes. The intuitive, formal and practical foundations by graph rewrite systems, however, encourage much to refine and finally apply the approach given in this contribution.

7   Conclusions and Future Work

A new approach for supporting the software engineer during the software design at early specification stages has been proposed in this paper. The idea is to specify not all parts of the system to be designed in a concrete manner but, instead, to specify some of those in a very abstract way. The goal is to use these as search patterns in order to find corresponding concretely specified components which can then be substituted for the former ones. This intuitive understanding could be underpinned by a formalization of this process by graphs and graph rewrite rules according to the algebraic double pushout approach. Finally, also the practical applicability of the approach could be shown and, even, an idea of automation could be sketched very coarsely.

The approach presented in this contribution will be investigated much more deeply in future work, also a more complex case study should be considered. Further, the solution ideas sketched in chapter 6 wrt the problems concerning similarities between software components will be elaborated much more precisely. We have used a specification expressed as an UML class diagram for demonstration. Beyond the semantics of UML class diagrams, however, one additionally could use other types of diagrams offered by UML for specifying software components in a more detailed way. Through a corresponding formalization by graphs and graph rewrite rules one can expect correct specification steps as well within an even more complex software design process.

The approach presented in this contribution to us seems to be very promising wrt a more understandable, correct and comfortable software design process.

Acknowledgments

8   References

[1]   Arefi, F., Milani, M., and Stary, Ch., 1991, "Towards Customized User Interface Design Environments", Journal of Visual Languages and Computing, IEEE, pp. 146-151.

[2]   Bergmann, R. and Eisenecker, U., 1995, "Case-Based Reasoning for Supporting the Reuse of Object-Oriented Software: A Case Study", *Proceedings der 3. Deutschen Expertensystemtagung*, XPS-95, Infix-Verlag, pp. 152-169 (in German).

[3]   Booch, G., 1994, "Object-Oriented Analysis and Design with Applications", Benjamin Cummings, Redwood City.

[4]   Booch, G., Jacobsen, I., and Rumbaugh, G., eds., 1997A, "UML Summary (Version 1.1)", Rational Cooperation, Santa Clara, `http://www.rational.com`.

[5]   Booch, G., Jacobsen, I., and Rumbaugh, G., eds., 1997B, "UML Semantics (Version 1.1)", Rational Cooperation, Santa Clara, `http://www.rational.com`.

[6]   Engels, G., Gall, R., Nagl, M., and Schäfer, W., 1983, "Software Specification Using Graph Grammars", Osnabrücker Schriften zur Mathematik, Reihe Informatik, Dept. of Mathematics, University of Osnabrück, No. 9.

[7]   Engels, G. and Schäfer, W., 1989, "Programmentwicklungsumgebungen - Konzepte und Realisierung", Teubner.

[8]   Freund, R., Haberstroh, B., and Stary, Ch., 1992, "Applying Graph Grammars for Task-Oriented User Interface Development", Proceedings IEEE Conference on Computing and Information ICCI'92, W.W.Koczkodaj et.~al., eds., pp. 389-392.

[9]   Goedicke, M., 1993, "On the Structure of Software Description Languages: A Component Oriented View", Habilitation Thesis, Technical Report No. 473/1993, Dept. of Computer Science, University of Dortmund.

[10]   Goedicke, M., Sucrow, B.E., 1996, "Towards a Formal Specification Method for Graphical User Interfaces Using Modularized Graph Grammars", *Proceedings of the Eighth International Workshop on Software Specification and Design*, IEEE Computer Society, IEEE Computer Society Press, March, 22-23; Schloss Velen, Germany, pp. 56-65.

[11]   Gogolla, M. and Parisi-Presicce, F., 1998, "State Diagrams in UML - A Formal Semantics using Graph Transformation", *Proceedings ICSE'98 Workshop on Precise Semantics of Modeling Techniques (PSMT'98)*, Broy, M., Coleman, D., Maibaum, T., and Rumpe, B., Eds., Technical University of Munich, Technical Report TUM-I9803, pp. 55-72.

[12]   Jacobsen, I., Christerson, M., Jonsson, P., and Overgaard, G.G., 1992, "Object-Oriented Software Engineering", Addison-Wesley.

[13]   Miller, G.A., Beckwith, R., Fellbaum, Ch., Gross, D., and Miller, K.J., 1990, "Introduction to WordNet: an on-line lexical database." *International Journal of Lexicography 3 (4)*, pp. 235 - 244.

[14]   Nowak, U., Pöhle, U., Roitzsch, R., and Sucrow, B.E., 1996, "Formal Specification of the ZIB-GUI Using Graph Grammmars" (in German), *Workshop ''Software Engineering im Scientific Computing'*, Hamburg, Germany, June 6-8, 1995, Mackens, W. and Rump, S.M., eds., Vieweg, pp. 290-296.

[15]   Richter, M., 1995, "Class Structure Graphs for Object-Oriented Design and Implementation", Technical Report, ifi-95.01, University of Zurich.

[16]   Rozenberg, G., ed., 1997, "Handbook of Graph Grammars and Computing by Graph Transformation", World Scientific, vol. 1.

[17]   Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., 1991, "Object-Oriented Modeling and Design", Prentice Hall, Englewood Cliffs.

[18]   Spanoudakis, G. and Constantopoulos, P., 1996, "Analogical Reuse of Requirements Specifications: A Computational Model", *Applied Artificial Intelligence: An International Journal*, Vol. 10, No. 4, pp. 281-306.

[19]   Sucrow, B.E., 1996A, "Formal Specification of Graphical User Interfaces Using Graph Grammars" (in German), *Workshop ''Software Engineering im Scientific Computing'*, Hamburg, Germany, June 6-8, 1995, Mackens, W. and Rump, S.M. , Eds., Vieweg, 279-289.

[20]   Sucrow, B.E., 1996B, "Towards an Integration of Software-Ergonomic Aspects in Formal Specifications of Graphical User Interfaces", *Proceedings of the Second World Conference on Integrated Design & Process Technology*, Vol. 1, Society for Design and Process Science, Austin, Texas, December 1-4, pp. 194-201.

[21]   Sucrow, B.E., 1997, "Formal Specification of Human-Computer Interaction by Graph Grammars under Consideration of Information Resources", *Proceedings of the 1997 Automated Software Engineering Conference (ASE'97)*, IEEE Computer Society, November 1-5, pp. 28-35.

[22]   Sucrow, B.E., 1998A, "On Integrating Software-Ergonomic Aspects in the Specification Process of Graphical User Interfaces", Accepted for publication in *Transactions of the SDPS Journal of Integrated Design and Process Science*, Society for Design and Process Science, IEEE Computer Society Press, 1998.

[23]   Sucrow, B.E., 1998B, "Refining Formal Specifications of Human-Computer Interaction by Graph Rewrite Rules", *Fundamental Approaches of Software Engineering* , Egidio Astesiano (Ed.), First International Conference, FASE'98, Held as Part of the of the Joint European Conferences on Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March/April 1998, Proceedings.