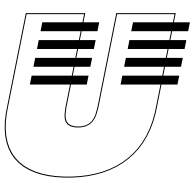


UNIVERSITÄT
KOBLENZ · LANDAU

Meta-CASE in Practice: a Case for KOGGE

Jürgen Ebert, Roger Süttenbach,
Ingar Uhe

22/96

Fachberichte
INFORMATIK

Universität Koblenz-Landau
Institut für Informatik, Rheinau 1, D-56075 Koblenz
E-mail: researchreports@infko.uni-koblenz.de,
WWW: http://www.uni-koblenz.de/universitaet/fb/fb4/

Meta-CASE in Practice: a Case for KOGGE

Jürgen Ebert, Roger Süttenbach, Ingar Uhe
University of Koblenz-Landau
Institute for Software Technology
{ebert, sbach, uhe}@informatik.uni-koblenz.de

Abstract

Meta-CASE tools are used to generate CASE tools; KOGGE is such a meta-CASE system. Two of KOGGE's main objectives are adaptability and flexibility which address the growing need for problem specific solutions. As an example it is shown how a CASE tool for the object-oriented method BON – BONsai – was constructed using KOGGE.

Keywords: Meta-CASE, CASE, declarative modeling, software engineering environments, BON

1 Introduction

CASE (Computer Aided Software Engineering) was one of the buzzwords of the late eighties. However, use of CASE tools in practice showed that they were not applied as widely as expected. One of the reasons for this is that there are so many different methods, dialects of methods, heterogeneous environments and continuously changing requirement. Thus, most CASE tools do not fit their specific context exactly. The inflexibility of the existing CASE systems demonstrated that there was a need to build systems which can be adjusted to the individual situation.

Meta-CASE tools – tools which are used to build CASE tools – solve this problem. There are two different kinds of meta-CASE systems. The first type offers a predefined set of supported notations which can be used. System like this are e.g. Toolframe [DK95] and Paradigm+ [P94].

Generators of CASE tools fit into the second category. They offer the possibility to produce completely new CASE tools. Metaview [Fi94] and MetaEdit [KLR96] are examples of this approach. In Koblenz, the meta-CASE system *KOGGE* [EC94][Me94] (KOblenz Generator for Graphical design Environments) has been developed to generate CASE tools which support visual languages and methods. Main objectives are adaptability and flexibility based on a declarative and graph-based approach.

In this paper the meta-CASE system KOGGE will be described. In order to illustrate the KOGGE approach it will be shown how KOGGE was used to implement a CASE tool supporting the object-oriented method BON.

The paper is organized as follows: section 2 sketches BON and the generated tool BONsai. The ideas of KOGGE and the construction process of a CASE tool are subject of section 3. Section 4 focusses on aspects of adaptability, and section 5 contains some concluding remarks.

2 BON and BONsai

In early 1995 the University of Dortmund was looking for a *CASE tool* which supports the *Business Object Notation* (BON). BON had been chosen to be used in a six-weeks software lab [B93]. Since such a lab has certain requirements with regard to a CASE tool and the method chosen was quite new, no appropriate tool for BON was found on the market.

Therefore, it was decided to use the meta-CASE system KOGGE to generate the tool needed. In less than three months a specially designed tool – named BONsai – was specified and tested. The first version was sucessfully used in the fall of 1995. At present, the third version of BONsai is being employed and it will be used with further extensions in 1997.

To introduce the way KOGGE tools work, a short description of the BON method is followed by a presentation of how the tools for BON – BONsai I, BONsai II and BONsai III – support the method.

2.1 BON

The object-oriented method BON ([WN95]) by Kim Waldén and Jean-Marc Nerson was influenced by the Eiffel school of thought. BON is guided by three main principles – *seamlessness*, *reversibility* and *software contracting*. Seamlessness and reversibility refer to the possibility of seamless transition from problem domain requirements, via system design to executable code, and vice versa. Software contracting reflects the development of software as a series of documented decisions which are realized by assertions. Assertions are expressed by pre- and postconditions for single operations or by invariants for all operations of a class.

The main visual elements of BON are *clusters*, *classes* and *relationships*. A cluster, which describes a group of related classes (or clusters, respectively), is represented graphically by a box with rounded corners and a class is represented graphically by an ellipse. The relationships are drawn as single or double lines with arrows. For each kind of relationship – inheritance, aggregation and association – specific arrow symbols are used.

A BON specification consists of varios kinds of documents: the *static architecture*, the *class interface* and the *charts*. A static architecture is a

diagram representing the class structure of a system which is a graph whose nodes are classes or clusters and whose arcs are relationships among them. A chart is an informal description of a class or a cluster whereas a class interface is a definition of a certain class describing its signature and its assertions.

2.2 BONsai

The CASE tool BONsai consists of *editors* for the static architecture, the class interfaces and the charts. Figure 1 shows the editor for the static architecture. It offers the creation and deletion of the graphical items used, automatic drawing of relationships between the symbols and a number of supporting functions such as zooming, enabling a grid and different views of the system.

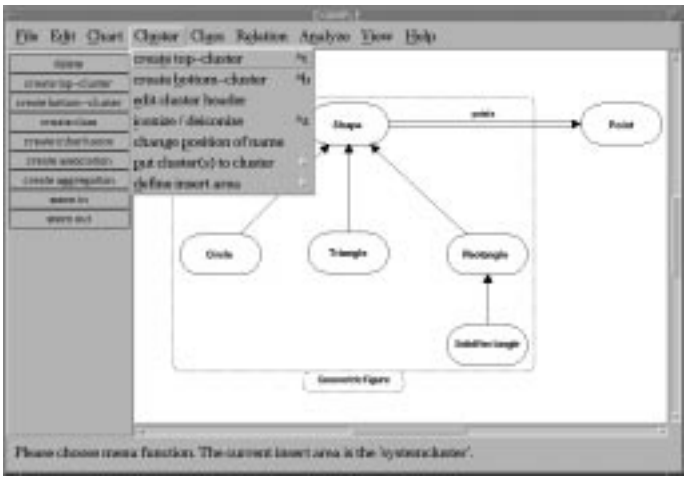


Figure 1: Editor for the static architecture in BONsai III

But of course, BONsai is not only an editing facility for the most important document types of the BON method as described above. Additionally, it supports the user in the process of creating correct models according to BON through *explicit* and *implicit constraints*.

Explicit contraints are tests that the user directly invokes by choosing the corresponding menu item whereas implicit contraints are permanently checked (and enforced) by the tool. An example for the latter is single inheritance which is mandatory in BONsai III. It supports this constraint by prohibiting the creation of an inheritance relation which would result into a model with multiple inheritance. On the other hand, checking uniqueness of the root class – the starting class of a model– is implemented as an explicit constraint.

In Dortmund, BONsai is integrated into a software development environment which covers the whole development cycle. Thus, in addition to support the user in the modeling process BONsai also offers the generation of code.

Although the preferred language for BON is Eiffel [M93], the code generated by BONsai is not restricted to a single target language. Content of the models is translated into a textual language which can be transformed into code frames of different programming languages. In BONsai I this was Eiffel while BONsai II and III generate code frames for BETA [DD96].

3 KOGGE

KOGGE's aim is to support the generation of tools for visual languages such as BON. KOGGE tools have also been generated for dataflow diagrams, entity relationship diagrams, state charts and others.

The basic idea is the following: instead of programming every single tool for all different methods, there is one *base system* – which is the same for all tools – and a specific *tool description* – which is unique. The tool description is interpreted by the base system, so that the combination of both of them results in a concrete tool. From the user's point of view the separation of the tool description from the base system is not visible. The users of a CASE tool generated with KOGGE just start one system.

The idea of '*table-driven*' tools is well-known from varios areas such as compiler-construction [ASU86].

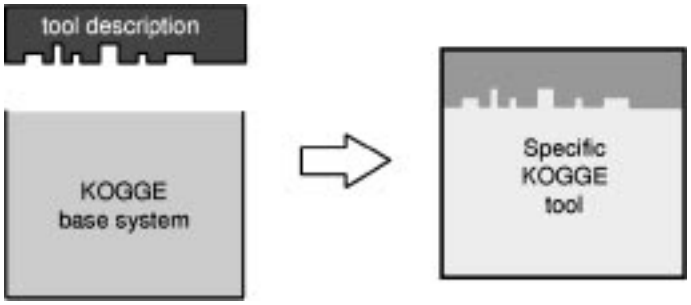


Figure 2: Structure of KOGGE tools

Using KOGGE in the development of BONsai meant: a suitable tool description had to be constructed. Like any other KOGGE tool BONsai works as shown in figure 2. The description for BON is interpreted by the base system, a screenshot of the resulting system was already shown in figure 1.

At first (section 3.1) we are going to describe how a tool description – actually the one for BONsai – is specified. This is done using another KOGGE tool called UrKOGGE. In contrast to constructing such a tool description for every new CASE tool, the base system (section 3.2) is always the same. Section 3.3 explains the UrKOGGE in more detail and the connection to the generator concept mentioned above.

3.1 Tool Description

A KOGGE tool description always consists of three parts, describing

1. the concepts of the method,
2. the structure of the menus, and
3. the interaction with the user.

The first item defines the structure of the repository, whereas the tool's behavior is determined by the interactions with the user connected to the menus. There are specific editors for all of the three components.

3.1.1 Describing the Concepts

Describing the concepts[1] of a method, or strictly speaking the concepts of its visual languages, means to describe what concepts are used, how they are combined and what context conditions must be preserved. In other words, the abstract syntax of the visual languages must be formalized.

In order to do this, one needs a formal basis to model and to implement the abstract syntax. In KOGGE, TGraphs [EF94] are used for this purpose which are the basis for the *EER/GRAL approach* of modeling and implementation with graphs comprehensively described in [EWD+96] and [CEW95]. Here, graph classes – which are sets of graphs – can be defined by *extended entity relationship (EER) descriptions* which are annotated by integrity conditions expressed in the *constraint language GRAL* (GRAph specification Language).

The EER/GRAL descriptions define the set of correct syntax graphs in the sense of the method, for example the set of correct BON models. This description is called a *metamodel*. This term refers to the fact that instances of this model are models themselves.

EER

An EER description uses *five different building blocks* for formalization – entity types, relationship types, attributes, generalizations, and aggregations. Figure 3 shows the corresponding graphical representation. Regarding the underlying graph approach an entity type defines a set of vertices whereas a relationship type defines a set of edges. An attribute adds additional information to vertices or edges. Generalization defines a hierarchy between vertex types whereas aggregation can be chosen to add structural information[2].

[1] Classes and clusters are examples for concepts in BON.

[2] Note that these elements are used in the BON metamodel, part of it is shown in figure 5.

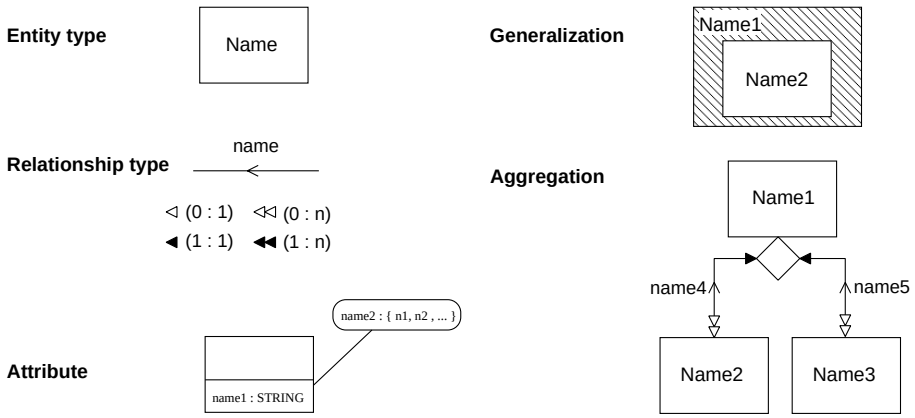


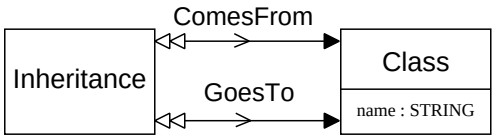
Figure 3: EER building blocks

GRAL

The $\mathcal{Z}$-like [S92] *assertion language GRAL* is used to denote those integrity conditions which cannot be expressed by the EER descriptions. These may be constraints on the values of the attributes of vertices and edges, the existence or non-existence of a certain path in a graph, the cardinality restrictions of vertex sets depending on attribute values, etc.

A GRAL assertion is a sequence of predicates which directly refer to the corresponding EER description. The syntax and the semantics of GRAL is formally described in [F96]. GRAL predicates can be tested efficiently.

Here, we do not describe GRAL in detail but figure 4 gives an example in the context of the BON method.



for G **in** *StaticDiagram* **assert**

SD1 : $\forall\, c : V_{Class} \bullet \neg\, (c\ (\leftharpoonup_{GoesTo} \bullet_{Inheritance} \rightharpoonup_{ComesFrom})^{+}\, c)$;

SD2 : $\forall\, c : V_{Class} \bullet \#\{c \leftharpoonup_{ComesFrom} \bullet_{Inheritance}\} \leq 1$;

Figure 4: Example: Part of the EER/GRAL description of BONsai

Inheritance and Class are concepts used in the BON method. A class can be related to zero or more inheritance relations whereas an inheritance relation must relate to exactly one start class and one end class. This is modeled by the ComesFrom and the GoesTo relationship and the corresponding cardinalities. Additionally, the constraint that no circles exist in the inheritance

hierarchy must be preserved, i.e. constructions like 'class A inherits from class B which inherits from class A' are not allowed in a BON model. This is expressed by the GRAL predicate SD1 below the EER description. SD2 prohibits the creation of multiple inheritance.

Editors

The EER/GRAL approach is the theoretical background to define the metamodel. To edit EER/GRAL descriptions for the concepts of a method two tools are used: A *visual ER Editor* and a *text editor for GRAL predicates*. The screenshot presented in figure 5 gives an impression which shows a part of the metamodel of BON. One can recognize the `GoesTo` and `ComesFrom` relationships of the small example above. The `Class` and `Inheritance` entity types are connected to them by their generalizations `ClassOrCluster` and `StaticLink`. The GRAL predicates are edited by a separate text editor which is not presented here.

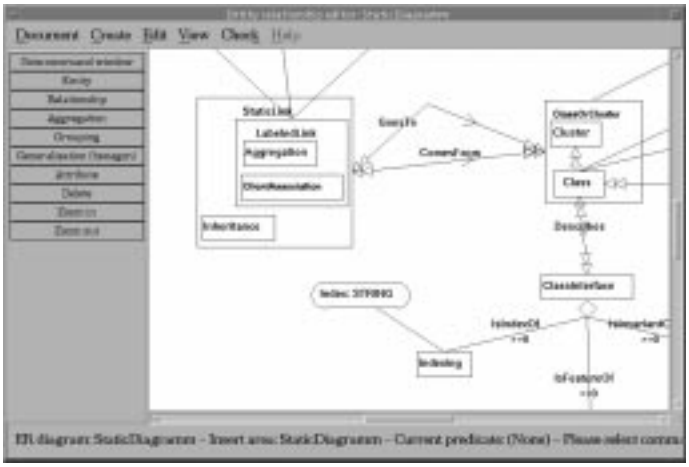


Figure 5: ER editor with a part of the metamodel for BON

Like all KOGGE tools the graphical editor supports the tool builder in different ways: It disposes the graphical symbols; it supports automatic drawing of relationships between entity types; it makes grids, overviews and zooming functions available; it checks the syntactical conditions of EER descriptions. This allows a comfortable and easy production of a metamodel[3].

3.1.2 Describing the Menus

In addition to describing the concepts of the method it is necessary to define the menu structure of the tool for a specific method. In KOGGE a (fairly

[3] Note that figure 5 is also a screenshot of a KOGGE tool. Thus, a KOGGE tool is used for editing tool descriptions for KOGGE tools as described in section 3.3 in more detail.

simple) visual language is used in order to define the menu structure.

The menu structure is a directed acyclic graph (dag). There are different types of nodes for *menus* and *menu items* and two special nodes: *root* and *panel*. Menu items or submenus are subordinated to one (or possibly multiple) menus by edges called connectors. Menus which are not submenus of any other menu have to be connected to root which serves as the parent of all menus on the top level. Menu items can – in addition to being connected to a menu – also be connected to the panel node which has the same function as toolbar buttons in other tools and can be seen in the left part of the screenshots. And finally, menu items have a few properties such as being enabled/disabled which can be set.

Editor

The creation of menus is also supported by an editor which offers operations to create, delete and edit all relevant items. Being a KOGGE editor, it provides the same supporting operations like different views etc. as the other editors (section 3.1.1 and 3.1.3).

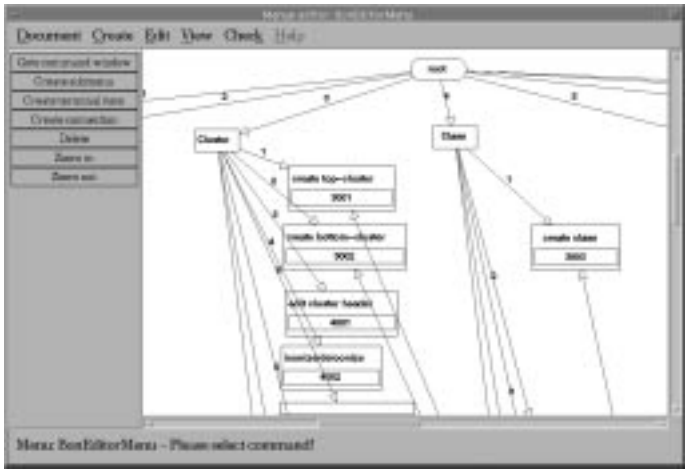


Figure 6: Menu editor with a part of the cluster and class menu

Figure 6 shows a part of the cluster menu. The `cluster menu` (figure 1) contains the menu items `create top-cluster`, `create bottom-cluster`, `edit cluster header`, `iconize/deiconize` etc. `Create top-cluster` and `create bottom-cluster` were also linked to the panel since they are frequently used operations. Most of the menu items have shortcuts like `Ctrl-t` for `create top-cluster`.

3.1.3 Describing the Interactions

The interactions between user and tool and the behaviour depending on these interactions are described by state charts [H87]. A state chart is a

graph whose nodes are – possibly compound – states and whose edges are transitions labeled with events and actions. In additon to state transition diagrams state charts possess a hierarchy which is obtained by superstates nesting other states; concurrency and broadcasting are further extensions of state charts.

The tool is always in a certain state, a state change is caused by an event triggering an action. In KOGGE events are defined by the menu items in the menu structure or are self-defined. Actions can be described by procedures which are written in a macro language called KOGGE-Modula. This macro language is similar to the programming language Modula-2 ([W85]) and allows an imperative programming style including the definition and evaluation of GRAL predicates. Additionally, the language is supported by a library that includes often used actions like file operations, a library of graphical symbols and graphical utility functions.

Editors

There is a graphical editor for state charts as well. This editor provides the tool builder with functionality in order to produce the state charts for the tool. It works similar to the other UrKOGGE editors. Additionally, the consistency of the state charts is checked, for example, that at most one start state exists in a superstate. Figure 7 presents the top level of the state charts used in BONsai.

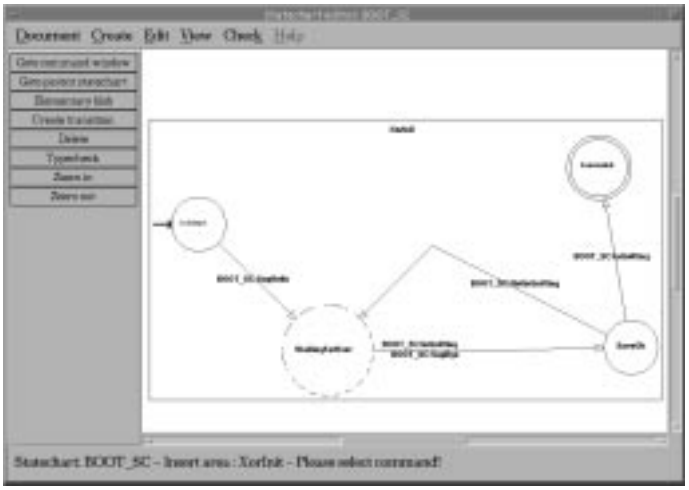


Figure 7: State chart editor with the top-level state chart of BONsai

In order to produce the procedures in KOGGE-Modula a simple text editor is given by default or one can use other text editors which are available for the UNIX system (like NEdit shown in figure 8). The figure also gives an impression of what KOGGE-Modula procedures look like. The small procedure inside checks the uniqueness of the root class (page 3).

Figure 8: A KOGGE-Modula procedure for BONsai

3.2 Base System

The base system is the static part of the KOGGE system (figure 2). It does not need to be changed or enhanced in the process of specifying a specific KOGGE tool. The base system can be divided in two parts: the *interpreters* and the *graphical modules*.

The interpreters [C96] are needed to interpret the tool description and thus produce the specific tool. Corresponding to the tool description there are three interpreters: the ER interpreter, the menu interpreter and the state chart interpreter. The ER interpreter is used to preserve the consistency of the tool with respect to its EER/GRAL description. The menu interpreter takes care of assembling the menus and the windows of the tool. And the state chart interpreter controls the tool depending on the underlying state chart descriptions and the user actions.

The functions of the graphical modules [Me95] are used to visualize the tool and its components. They offer the user interface including components such as dialog boxes, file selection boxes etc. on the one hand and on the other hand they provide a library of symbols like circles, rectangles and lines which can be used. The kernel of the graphical modules is VarioCAD [DRB93] which is originally a CAD system. In addition to graphical functions such as zooming and alignment the graphical modules also provide operations which preserve the structure of the presentation. For example, a symbol which is logically positioned inside another symbol is moved simultaneously with the outer symbol.

The base system is implemented in C++, employs X-Windows and is currently available for Solaris, SunOS and Linux.

3.3 UrKOGGE and Bootstrapping

In section 3.1 we demonstrated how a CASE tool description for BON was produced using the three editors: ER editor, state chart editor and menu

editor. These three editors are also KOGGE editors and are parts of the tool called *UrKOGGE*. UrKOGGE is a KOGGE tool itself, that means it consists of a tool description and the base system as any other KOGGE tool. There is only one difference: instead of manipulating models of a certain CASE method, UrKOGGE manipulates tool descriptions, for example the BONsai description as shown in figure 9.

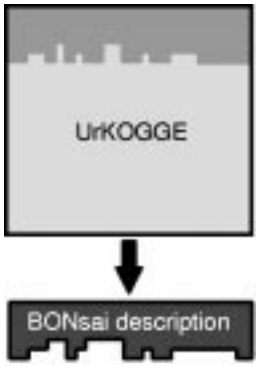


Figure 9: Using the UrKOGGE

In UrKOGGE, tool descriptions are normal data, both are handled in the same way. This is possible due to the underlying structure which allows a unifying mapping of all information to a single representation – namely *graphs*. Graphs are created, stored and manipulated using the GraLab software, a C++ class library for graph-based software development [DEL95].

As a matter of fact, the tool description for the UrKOGGE can be edited in exactly the same way as the BONsai description. Of course, the very first version of the UrKOGGE description had to be generated from a text document. This rudimentary version is on the one hand used to control the base system, and on the other hand it can be loaded as data which can be manipulated (as shown in figure 10).

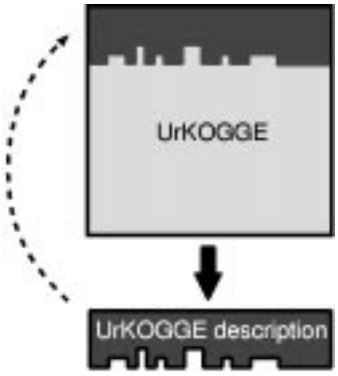


Figure 10: Bootstrapping of the UrKOGGE

The person who is specifying the tool can enhance the data – the UrKOGGE

description – and add further functions to it. This modified UrKOGGE description can then be used as a new controling tool description. This process of enhancing and improving the UrKOGGE description can be iterated.

Summing up, one can say this strategy of being able to manipulate/-modify/enhance the UrKOGGE specification in the same manner as the tool descriptions for other tools has several advantages. The main advantage is that we can change the UrKOGGE easily without having effects on other existing KOGGE tools.

4 Adaptability

The use of CASE tools in the past, or the disappointment about the use of CASE tools in practice, has shown that flexibility and adaptability are very important qualities of any of these systems. There is a growing need for CASE tools to be adjusted to specific contexts, dialects of methods and changing requirements, to name just a few aspects. The specification of the CASE tool and its functionality, like it is done in KOGGE, makes it easy to meet these requirements which we will illustrate by a few more examples from the CASE tool BONsai.

The *context* or the situation for which a tool is used is important for the design of this tool. Since the department of computer science in Dortmund needed a CASE tool for a six-weeks software lab it was important that the resulting tool was simple enough that it could be learned easily, yet it had to have all the functionality to serve its purpose. The circumstances in which the tool was to be used lead to other customized features. For example, the tool should not have an `open file`-menu item since the tool is embedded in a special environment.

CASE tools are subject to *changing requirements*. When a system is used practically, one finds out which parts are really used and necessary and which further functionality has possibly to be added. BONsai I, for example, allows multiple inheritance, while BONsai II and III only permit single inheritance. This was due to the fact that Eiffel code was generate in the first version and BETA in the further ones.

Most *methods* used nowadays are not exhaustively explained or even inconsistent. Thus, people building tools have to make decisions what certain concepts really mean. Our approach makes it easy to change these interpretations of method features in case they were wrong or not useful.

The ability of adaptability was a main objective in designing KOGGE. A KOGGE tool is generated by a mostly declarative description in place of operational programming. This declarative style leads to a higher level of abstraction which makes adaptability easy. For example, the shifting from multiple to single inheritance was achieved only by adding the second constraint SD2, already shown in figure 4, to the EER/GRAL description for

BONsai I:

$$\mathsf{SD2}: \quad \forall\, c : V_{Class} \bullet \#\{c \leftharpoonup_{ComesFrom} \bullet_{Inheritance}\} \leq 1;$$

No further modifications were needed.

5 Conclusion

Meta-CASE tools are needed to generate problem and context specific CASE tools. Our approach to meta-CASE – the system KOGGE – is suited to meet this demand because it is formally *well-founded* and was sucessfully *used in practice* (each of the three versions of BONsai was used by 80-100 students). BONsai is not the only CASE tool specified, we also have one for dataflow diagrams [D96], the UrKOGGE itself and a tool to support the evaluation of software is under construction.

Note that we did not describe how one can produce *metamodels* for the languages used in these tools. This is a different research area which needs profound discussions, examples are given in [ES96a] und [ES96b].

All in all one can say that generating CASE tools with KOGGE can – due to the *high abstraction level* of the description – be achieved in a short amount of time and the resulting tools are easily modified as shown in the previous sections. Moreover the tool to specify other CASE tools itself can be changed since its tool control can be manipulated in the same manner. This flexibility is achieved because graphs are used for either one: for tool descriptions and for data representation. At last, CASE tools developed with KOGGE all have a common look and feel because they all rely on the same base system.

But even though KOGGE tools have been used in practice, there is still a lot of room for improvement. As a research prototype KOGGE has always been focused to certain research areas. Thus, it does not support some important features for large scale applications, like multi-user facilities and version management, but concentrates mainly on adaptability in a single user environment. As the BONsai experience shows, KOGGE works quite well with regard to this.

Acknowledgment: The authors express their thanks to Manfred Rosendahl for supervising the development of the graphical modules, the crew in the computer science department in Dortmund especially Thomas Biedassek, the former project members Martin Carstensen and Albrecht Meißner and numerous students for their work for the project and Angelika Franzke for thoroughly revising the paper.

References

[ASU86] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.; *Compilers principles, techniques and tools.* Reading, Mass.: Addison-Wesley, 1986.

[B93] Biedassek, Thomas; *Software in der Ausbildung.* Projekt OPHELIA für die Ausbildung in Software-Technik. In: UNIX-open – Reihe Praxis, 8/93, S. 76.

[C96] Carstensen, Martin; *Konzept eines CASE-Generators für CASE-Umgebungen.* Koblenz: erscheint als Dissertation, 1996.

[CEW95] Carstensen, Martin; Ebert, Jürgen; Winter, Andreas; *Entity-Relationship-Diagramme und Graphklassen.* Koblenz: Universität Koblenz-Landau, erscheint als Fachbericht Informatik, 1996.

[DK95] Däberitz D., Kelter, U.; *Rapid Prototyping of Graphical Editors in an Open SDE.* In: Proc. 7th Conf. on Software Engineering Environments (SEE '95), p. 61-72. Noordwijkerhout: IEEE Computer Society Press, 1995.

[DEL95] Dahm, Peter; Ebert, Jürgen; Litauer, Christoph; *Das EMS-Graphenlabor V3.0.* Koblenz: Universität Koblenz-Landau, Projektbericht, 1995.

[DD96] Doberkat, Ernst-Erich; Dissmann, Stefan; *Einführung in die Programmierung mit BETA.* Bonn: Addison-Wesley, 1996.

[D96] Drüke, Maurice; *Dokumentation für den Datenflußdiagram-Editor.* Koblenz: Institut für Softwaretechnik, Studienarbeit, 1996.

[DRB93] Du, Chun; Rosendahl, Manfred; Berling, Roland; *Variation of Geometry and Parametric Design.* In: Proc. 3rd. international conference on CAD and computer graphics, Beijing, Aug. 23-26, 1993, p. 400-405, international academic publishers, 1993.

[EC94] Ebert, Jürgen; Carstensen, Martin; *Ansatz und Architektur von KOGGE.* Koblenz: Universität Koblenz-Landau, Institut für Softwaretechnik, Interner Projektbericht 2/94, 1994.

[EWD+96] Ebert, Jürgen; Winter, Andreas; Dahm, Peter; Franzke, Angelika; Süttenbach, Roger; *Graph Based Modeling and Implementation with EER/GRAL.* In: B. Thalheim [Ed.]; 15th International Conference on Conceptual Modeling (ER'96), Lecture Notes In Computer Science, Proceedings, LNCS 1157. Berlin: Springer, 1996.

[EF94] Ebert, Jürgen; Franzke, Angelika; *A Declarative Approach to Graph Based Modeling.* In: Mayr, E.; Schmidt, G.; Tinhofer, G. [Eds.]; Graphtheoretic Concepts in Computer Science, Lecture Notes in Computer Science, LNCS 903, p. 38-50. Berlin: Springer, 1995.

[ES96a] Ebert, Jürgen; Süttenbach, Roger; *An OMT Metamodel.* Koblenz: Universität Koblenz-Landau, erscheint als Fachbericht Informatik, 1996.

[ES96b] Ebert, Jürgen; Süttenbach, Roger; *A Booch Metamodel.* Koblenz: Universität Koblenz-Landau, erscheint als Fachbericht Informatik, 1996.

[Fi94] Findeisen, Piotr; *The Metaview System.* Edmonton: Department of Computing Science, University of Alberta, 1994.

[F96] Franzke, Angelika; *GRAL: A Reference Manual.* Koblenz: Universität Koblenz-Landau, Fachbericht Informatik 11/96, 1996.

[H87] Harel, David. *Statecharts: a visual formalism for complex systems.* Science of Computer Programming 8 (1987, 3), p. 231-274.

[KLR96] Kelly, Steven; Lyytinen, Kalle; Rossi, Matti; *MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment.* In: Constantopoulos, P.; Mylopoulos, J.; Vassiliou, Y. [Eds.]; Advanced Information System Engineering, Lecture Notes In Computer Science, LNCS 1080, Berlin: Springer, 1996.

[KU95] Kölzer, Anke; Uhe, Ingar; *Benutzerhandbuch für das KOGGE-Tool BONsai III.* Koblenz: Institut für Softwaretechnik, Interner Projektbericht 4/96, 1996.

[Me94] Meißner, Albrecht; *Die Präsentationskomponente von KOGGE.* Koblenz: Universität Koblenz-Landau, Institut für Softwaretechnik, Interner Projektbericht 1/94, 1994.

[Me95] Meissner, Albrecht; *GRABE – Eine objekt-orientierte Sprache zur Spezifikation von Symbolen in interaktiven graphischen Editoren.* Koblenz: Fölbach, Dissertation, 1995.

[M93] Meyer, Bertrand; *Eiffel – the language.* New York: Prentice Hall, 1993.

[P94] ProtoSoft Inc. *Paradigm +/Cadre Edition Reference Manual.* Providence: Protosoft, 1994.

[S92] Spivey, J.M.; *The $\mathcal{Z}$ Notation.* A Reference Manual. Prentice Hall: Hemel Hempstead, 1992[2].

[WN95] Waldén, Kim and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture.* Analysis and Design of Reliable Systems. Prentice Hall: Englewood Cliffs, 1995.

[W85] Wirth, Niklaus; *Programming in Modula-2.* Berlin: Springer, 1985[3].

Available Research Reports (since 1994):

1996

22/96 *Jürgen Ebert, Roger Süttenbach, Ingar Uhe.* Meta-CASE in Practice: a Case for KOGGE.

21/96 *Harro Wimmel, Lutz Priese.* Algebraic Characterization of Petri Net Pomset Semantics.

20/96 *Wenjin Lu.* Minimal Model Generation Based on E-Hyper Tableaux.

19/96 *Frieder Stolzenburg.* A Flexible System for Constraint Disjunctive Logic Programming.

18/96 *Ilkka Niemelä (Ed.).* Proceedings of the ECAI'96 Workshop on Integrating Nonmonotonicity into Automated Reasoning Systems.

17/96 *Jürgen Dix, Luis Moniz Pereira, Teodor Przymusinski.* Non-monotonic Extensions of Logic Programming: Theory, Implementation and Applications (Proceedings of the JICSLP '96 Postconference Workshop W1).

16/96 *Chandrabose Aravindan.* DisLoP: A Disjunctive Logic Programming System Based on PROTEIN Theorem Prover.

15/96 *Jürgen Dix, Gerhard Brewka.* Knowledge Representation with Logic Programs.

14/96 *Harro Wimmel, Lutz Priese.* An Application of Compositional Petri Net Semantics.

13/96 *Peter Baumgartner, Ulrich Furbach.* Hyper Tableaux and Disjunctive Logic Programming.

12/96 *Klaus Zitzmann.* Physically Based Volume Rendering of Gaseous Objects.

11/96 *J. Ebert, A. Winter, P. Dahm, A. Franzke, R. Süttenbach.* Graph Based Modeling and Implementation with EER/GRAL.

10/96 *Angelika Franzke.* Querying Graph Structures with G^2QL.

9/96 *Chandrabose Aravindan.* An abductive framework for negation in disjunctive logic programming.

8/96 *Peter Baumgartner, Ulrich Furbach, Ilkka Niemelä .* Hyper Tableaux.

7/96 *Ilkka Niemelä, Patrik Simons.* Efficient Implementation of the Well-founded and Stable Model Semantics.

6/96 *Ilkka Niemelä .* Implementing Circumscription Using a Tableau Method.

5/96 *Ilkka Niemelä .* A Tableau Calculus for Minimal Model Reasoning.

4/96 *Stefan Brass, Jürgen Dix, Teodor. C. Przymusinski.* Characterizations and Implementation of Static Semantics of Disjunctive Programs.

3/96 *Jürgen Ebert, Manfred Kamp, Andreas Winter.* Generic Support for Understanding Heterogeneous Software.

2/96 *Stefan Brass, Jürgen Dix, Ilkka Niemelä, Teodor. C. Przymusinski.* A Comparison of STATIC Semantics with D-WFS.

1/96 *J. Ebert (Hrsg.).* Alternative Konzepte für Sprachen und Rechner, Bad Honnef 1995.

1995

21/95 *J. Dix and U. Furbach.* Logisches Programmieren mit Negation und Disjunktion.

20/95 *L. Priese, H. Wimmel.* On Some Compositional Petri Net Semantics.

19/95 *J. Ebert, G. Engels.* Specification of Object Life Cycle Definitions.

18/95 *J. Dix, D. Gottlob, V. Marek.* Reducing Disjunctive to Non-Disjunctive Semantics by Shift-Operations.

17/95 *P. Baumgartner, J. Dix, U. Furbach, D. Schäfer, F. Stolzenburg.* Deduktion und Logisches Programmieren.

16/95 *Doris Nolte, Lutz Priese.* Abstract Fairness and Semantics.

15/95 *Volker Rehrmann (Hrsg.).* 1. Workshop Farbbildverarbeitung.

14/95 *Frieder Stolzenburg, Bernd Thomas.* Analysing Rule Sets for the Calculation of Banking Fees by a Theorem Prover with Constraints.

13/95 *Frieder Stolzenburg.* Membership-Constraints and Complexity in Logic Programming with Sets.

12/95 *Stefan Brass, Jürgen Dix.* D-WFS: A Confluent Calculus and an Equivalent Characterization..

11/95 *Thomas Marx.* NetCASE — A Petri Net based Method for Database Application Design and Generation.

10/95 *Kurt Lautenbach, Hanno Ridder.* A Completion of the S-invariance Technique by means of Fixed Point Algorithms.

9/95 *Christian Fahrner, Thomas Marx, Stephan Philippi.* Integration of Integrity Constraints into Object-Oriented Database Schema according to ODMG-93.

8/95 *Christoph Steigner, Andreas Weihrauch.* Modelling Timeouts in Protocol Design..

7/95 *Jürgen Ebert, Gottfried Vossen.* I-Serializability: Generalized Correctness for Transaction-Based Environments.

6/95 *P. Baumgartner, S. Brüning.* A Disjunctive Positive Refinement of Model Elimination and its Application to Subsumption Deletion.

5/95 *P. Baumgartner, J. Schumann.* Implementing Restart Model Elimination and Theory Model Elimination on top of SETHEO.

4/95 *Lutz Priese, Jens Klieber, Raimund Lakmann, Volker Rehrmann, Rainer Schian.* Echtzeit-Verkehrszeichenerkennung mit dem Color Structure Code — Ein Projektbericht.

3/95 *Lutz Priese.* A Class of Fully Abstract Semantics for Petri-Nets.

2/95 *P. Baumgartner, R. Hähnle, J. Posegga (Hrsg.).* 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods — Poster Session and Short Papers.

1/95 *P. Baumgartner, U. Furbach, F. Stolzenburg.* Model Elimination, Logic Programming and Computing Answers.

1994

18/94 *W. Hower, D. Haroud, Z. Ruttkay (Eds.).* Proceedings of the AID'94 workshop W9 on Constraint Processing in Computer-Aided Design.

17/94 *W. Hower.* Constraint satisfaction — algorithms and complexity analysis.

16/94 *S. Brass, J. Dix.* A Disjunctive Semantics Based on Unfolding and Bottom-Up Evaluation.

15/94 *S. Brass, J. Dix.* A Characterization of the Stable Semantics by Partial Evaluation.

14/94 *Michael Möhring.* Grundlagen der Prozeßmodellierung.

13/94 *D. Zöbel.* Program Transformations for Distributed Control Systems.

12/94 *Martin Volk, Michael Jung, Dirk Richarz, Arne Fitschen, Johannes Hubrich, Christian Lieske, Stefan Pieper, Hanno Ridder, Andreas Wagner.* GTU – A workbench for the development of natural language grammars.

11/94 *S. Brass, J. Dix.* A General Approach to Bottom-Up Computation of Disjunctive Semantics.

10/94 *P. Baumgartner, F. Stolzenburg.* Constraint Model Elimination and a PTTP Implementation.

9/94 *K.-E. Großpietsch, R. Hofestädt, C. Steigner (Hrsg.).* Workshop Parallele Datenverabeitung im Verbund von Hochleistungs-Workstations.

8/94 *Baumgartner, Bürckert, Comon, Frisch, Furbach, Murray, Petermann, Stickel (Hrsg.).* Theory Reasoning in Automated Deduction.

7/94 *E. Ntienjem.* A descriptive mode inference for normal logic programs.

6/94 *A. Winter, J. Ebert.* Ein Referenz-Schema zur Organisationsbeschreibung.

5/94 *F. Stolzenburg.* Membership-Constraints and Some Applications.

4/94 *J. Ebert (Hrsg.).* Alternative Konzepte für Sprachen und Rechner – Bad Honnef 1993.

3/94 *J. Ebert, A. Franzke.* A Declarative Approach to Graph Based Modeling.

2/94 *M. Dahr, K. Lautenbach, T. Marx, H. Ridder.* NET CASE: Towards a Petri Net Based Technique for the Development of Expert/Database Systems.

1/94 *U. Furbach.* Theory Reasoning in First Order Calculi.