Intelligent Software Synthesis in the DaCapo Environment

B. Steffen* B. Freitag A. Claßen T. Margaria U. Zukowski

Universität Passau
Germany

Abstract

The DaCapo software synthesis environment presented in this paper supports the *automatic construction of software* from existing components according to a (possibly incomplete) specification. Based on a specification language that uniformly combines taxonomic component specifications, interface conditions, and ordering constraints, our method adds a global view to conventional single component retrieval. Together with its semi-automatic archiving facility for the 'retrievable' inclusion of a satisfactory new software component into the software repository, our systems supports a full software lifecycle, as we illustrate along a user session.

1 Introduction

Systematic computer aided component reuse is essential to improve the productivity as well as the quality of software development. However, the complexity of the task requires to go beyond 'classical' library management tools, which are mainly based on single component retrieval, in order to sufficiently deal with the *global* dependencies between different software components. By their nature, such extensions are algorithmically very expensive. Thus only extremely carefully designed systems, which limit the exponential worst case behaviour to the 'unavoidable', are practically relevant. This requires the application specific structuring of the software repository together with a flexible specification language which allows users to formulate all the available knowledge without being forced to overspecify.

The DaCapo system supports a full software lifecycle satisfying these requirements for a 'transformational' setting by

- structuring the repository according to type constraints and semantic properties formulated in a taxonomic logic,
- providing a flexible specification language that captures *local* properties like taxonomic component descriptions and type conditions, and *global*

*Fakultät für Mathematik und Informatik, Universität Passau, D-94030 Passau (Germany), tel: +49 851 509-345, fax: +49 851 509-346, steffen@fmi.uni-passau.de

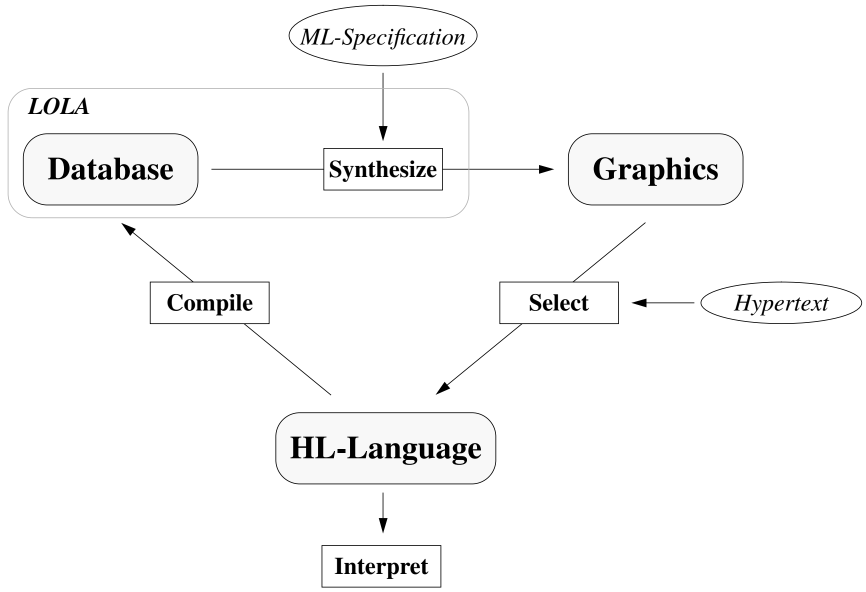

Figure 1: The DaCapo lifecycle

constraints fixing ordering, precedence, eventuality and conditional occurrence. This language implicitly encodes dynamic search strategies for sequential component compositions, which allows us to tailor the search effort according to the stage of the analysis.

- supporting the inclusion of a satisfactory new software component into the software repository by proposing an appropriately precomputed default taxonomic classification. These classifications are sufficient for future retrieval, but can also be modified by means of a hypertext system.

Figure 1 summarizes the architecture of our prototype implementation. Its database as well as its retrieval and synthesis algorithm are implemented on top of the deductive database system *LOLA* [FrKe94, FrSS91], and the graphical interface as well as the Hypertext browser are built on top of the TCL/TK graphics library [Oust94]. The target language is a high-level language, which is taylored for the combination of complex programs written in different programming languages. It is implemented in C++.

The development of the DaCapo system was influenced by industrial experience made with the hypertext-based software library system SEL [Frei94] that has been installed at BMW, Munich, Germany. The SEL system supports the classification and taxonomic grouping of components as well as the definition of typed links that allow a context dependent retrieval. However, neither type and signature information nor component composition have been considered, which are the essence of the DaCapo approach.

Current applications of our system range from the automatic construction of interfacing modules bridging the gap between the output type and the input type for a given sequence of software components, to the automatic synthesis of complete 'transformational' programs, which will be illustrated by configuring UNIX piping sequences for text and image processing (see Section 4). Other practically relevant applications that we are currently investigating concern

- semi-automatic intelligent network programming in a cooperation with Siemens Nixdorf (Munich),
- automatic composition of heterogeneous verification algorithms for hardware and software systems from basic transformation and analysis components (see [ClPS93, BuSt92, BuSt94]), and
- automatic generation of dataflow analysis algorithms from logic specifications for both finite and infinite state programs (see [Stef93, KnRS92, KnRS94a, KnRS94b]).

The paper is organized as follows: Section 2 introduces our UNIX application, Section 3 defines the metalevel specification language SLTL, Section 4 illustrates the full DaCapo lifecycle by means of a typical user session, Section 5 discusses the relations with other approaches, and Section 6 draws our conclusions.

2 Application: Synthesis of Complex UNIX Commands

We demonstrate the power of our approach by attacking a problem most UNIX users are confronted with: the synthesis of their own set of custom UNIX shell commands. The success of the UNIX operating system is in fact also due to its potential for composition-based software reuse: the UNIX shell can be viewed as a collection of reusable components that can be composed using the pipe mechanism. Let us consider text processing for illustration. Most UNIX environments offer the commands and applications summarized in Table 1, which we will use through the exemplary development in this paper.

The Models

UNIX commands and applications can be seen as transformations that take an input and produce the corresponding output. Thus they can abstractly be described as (*input-type*, `operation`, *output-type*). Accordingly, the `latex` application transforming a document *source.tex* into a device independent formatted document *source.dvi* is modelled as

$$(\mathit{latex\_source}, \mathtt{latex}, \mathit{dvi})$$

Similarly, the X-Windows display program `xdvi` which takes a *dvi*-file as input and displays it on the *screen* is described as

$$(\mathit{dvi}, \mathtt{xdvi}, \mathit{screen})$$

module name	input type	output type	description
`dvips`	*dvi*	*ps*	converts *dvi* to *ps*
	dvi/eps	*ps*	
`ghostview`	*ps*	*screen*	displays PostScript documents
`latex`	*ltx*	*dvi*	LaTeX-Compiler
	ltx/eps	*dvi/eps*	
`latex2html`	*ltx*	*html*	converts LaTeX source to *html*
`lpr`	*ps*	*printout*	prints documents
`more`	*ascii*	*screen*	displays plain text
`psnup-2`	*ps*	*ps*	formats multiple pages per sheet
`psselect`	*ps*	*ps*	selects pages from a document
`tex`	*tex*	*dvi*	TeX-Compiler
	tex/eps	*dvi/eps*	
`xdvi`	*dvi*	*screen*	TeX-Previewer
`xmosaic`	*html*	*screen*	World Wide Web browser

Table 1: The text processing module database

basic type	description
ascii	plain ASCII text
dvi	pure DVI file
dvi/eps	DVI file with embedded PostScript code
fig	vector graphic format
html	source file for www/xmosaic
ltx	pure LaTeX source
ltx/eps	LaTeX source with embedded PostScript code
pbm	portable bitmap graphic format
pgm	portable graymap graphic format
ppm	portable pixmap graphic format
printout	output type for print commands
ps	PostScript file
screen	output type for display commands
tex	pure TeX source
tex/eps	TeX source with embedded PostScript code

Table 2: The basic types

Table 1 and Table 2 summarize the commands and types we will use in this paper. Note that there can be more than one input/output combination for a command, thus allowing polymorphy.

Our goal is the specification and automatic synthesis of *legal program compositions*, i.e., of executable finite sequences of programs. Obviously, two programs can be sequentially combined iff they are *neighbours*, i.e. the output type of the first is compatible with the input type of the second. E.g. the `latex` and `xdvi` commands are neighbours, but `latex` and `ghostview` are not. Thus legal program compositions are simply sequential compositions of successively neighbouring programs, like e.g.

$(latex\_source, \texttt{display\_latex}, screen) = (latex\_source, \texttt{latex}, dvi, \texttt{xdvi}, screen)$

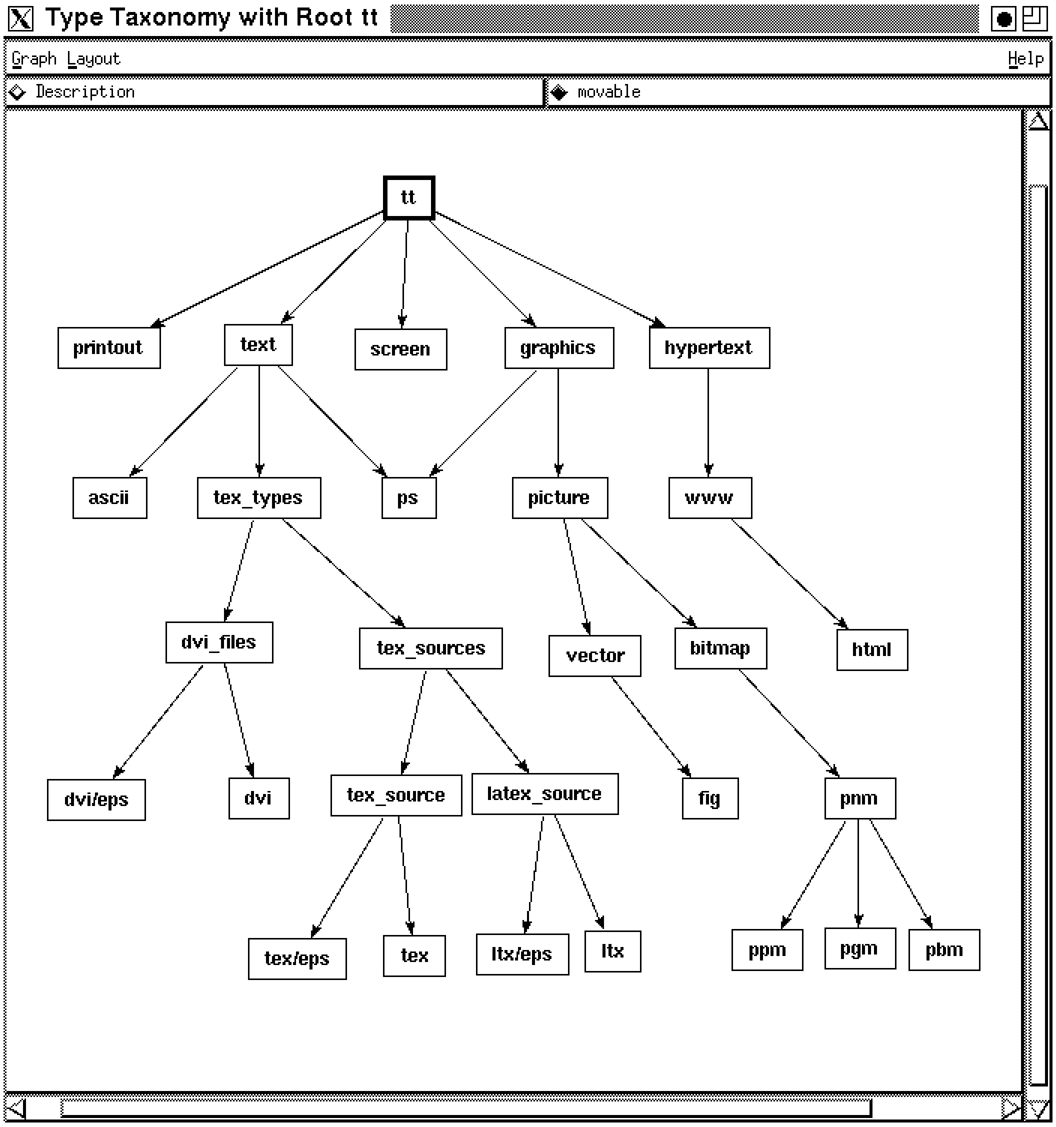

Figure 2: Hypertext browser - the type hierarchy

3 The Metalevel Specification Language

The DaCapo system is designed to synthesize the set of solutions to specifications given in terms of global and local constraints. These constraints can

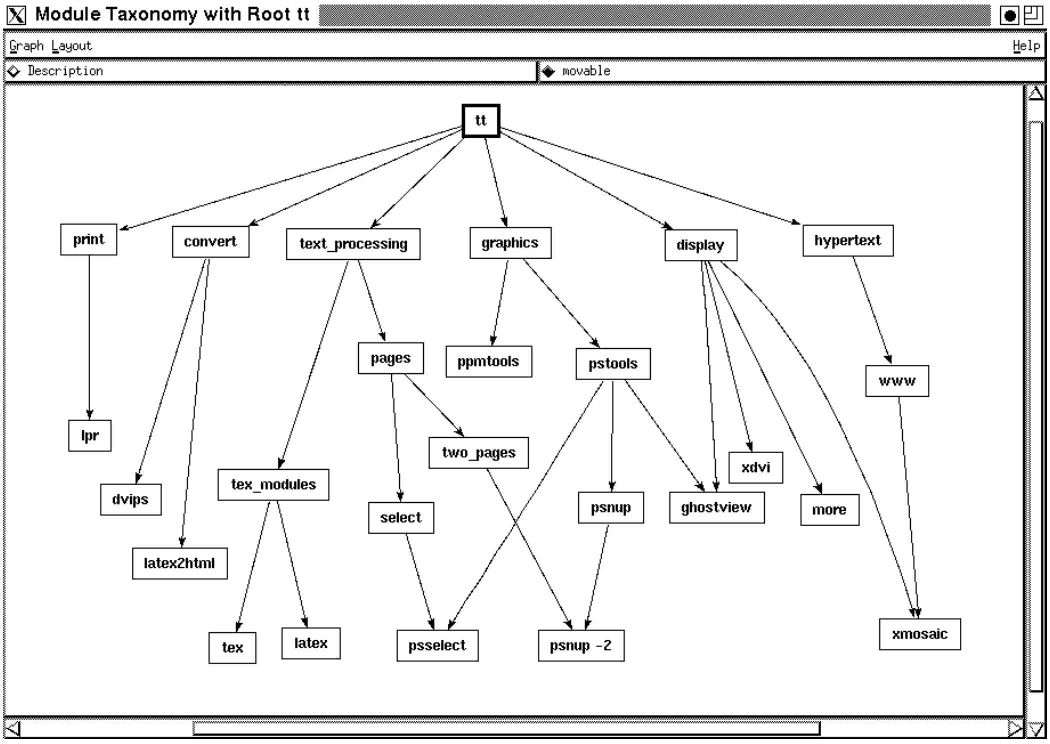

Figure 3: Hypertext browser - the module hierarchy

uniformly and elegantly be formulated within the temporal logic SLTL (Semantic Linear Time Logic), which combines *type*, *module*, and *ordering* constraints to a specification language with a coherent semantics. Thus more precisely, given an SLTL specification, the DaCapo system determines the set of all legal program compositions satisfying these constraints.

In practice, the set of all solutions is often very large. Thus it is convenient to define some default constraints in order to keep the set of delivered solutions manageable. In our system, *minimality* of the solutions is taken as such a default constraint. This choice reflects the desire for simple solutions and it does not affect the solvability of the specification.

The following two subsections present our languages for expressing *local* constraints capturing interfacing and module specifications, and *global* constraints restricting the ordering between modules.

3.1 Local Constraints

Local constraints are the selection criteria for single components. Currently they are based on a taxonomic classification of the repository components and the responsibility of an appropriate classification is left to the designer of the taxonomy. This choice does not impose any limitation, as our approach is parameterized in the local constraint mechanisms. In particular, the taxonomic classification of components described by Prieto-Diaz [Prie91, PrFr87], which can be found in most software library systems, could be easily integrated in our system.

Interface constraints describe the 'neighbourhood' of programs. In our UNIX application we consider type compatibility as the only criterion: Types are organized into a taxonomy according to their hierarchical classification (see Figure 2). A *type constraint* is formulated via simple logic formulas over the type taxonomy, and is interpreted in terms of sets of basic types. E.g. the formula type(*(text* or *html)* and not *ps*) is satisfied by the basic types

$$ascii, dvi, dvi/eps, html, ltx, ltx/eps, tex, tex/eps$$

Module constraints restrict the set of legal elementary programs which may appear in the solution. Like types, also the modules are organized in a taxonomic hierarchy (see Figure 3), and module constraints are represented by simple logic formulas over the module taxonomy that specify sets of elementary components. E.g. the formula module(`display` and not `hypertext`) is satisfied by the elementary commands

`ghostview`, `more`, `xdvi`

In general the full power of the deductive database language can be used to express local constraints. It is therefore possible to incorporate any sophisticated classification scheme and taxonomic retrieval into our system.

3.2 Global Constraints: The Temporal Aspect

Ordering constraints allow users to specify a relative ordering between (some of) the components that form a solution. Using a modal logic as a framework, we are able to extend the two local kinds of specifications to an elegant and uniform specification language for global constraints, which is defined as follows:[1]

Definition 3.1 (SLTL)

The syntactic structure of Semantic Linear Time Logic (SLTL) is given by:

$$\begin{aligned}\Phi ::= \;& \mathtt{tt} \mid \mathsf{type}(C_{Type}) \mid \mathsf{not}\ \Phi \mid \Phi\ \mathsf{and}\ \Phi \mid \Phi\ \mathsf{or}\ \Phi \mid \\ & \langle C_{Mod}\rangle\ \Phi \mid \mathsf{generally}\ \Phi \mid \Phi\ \mathsf{until}\ \Psi\end{aligned}$$

where C_{Type} and C_{Mod} represent type and module constraints, respectively. □

SLTL formulas are interpreted over the set of legal program compositions. Intuitively,

- `tt` is satisfied by every program composition.

[1]The specification language can be seen as an 'interpreted version' of Propositional Linear Temporal Logic (PLTL) [Emer90, SiCl85] or a linear time variant of the temporal Logic CTL [ClES86].

- type(C_{Type}) is satisfied by every program composition whose first component satisfies the type constraint C_{Type}[2].

- Conjunction, disjunction, and negation are interpreted in the usual fashion.

- <C_{Mod}> Φ is satisfied by program compositions whose first component satisfies C_{Mod} and whose *continuation*[3] satisfies Φ. In particular, <.> Φ is satisfied by every program composition whose continuation satisfies Φ.

- generally Φ requires that Φ is satisfied for every postfix[4] of the program composition. As we are dealing with finite paths only, certain properties of this kind will never hold. Still, this operator is useful to express global type constraints.

- Φ until Ψ expresses that the property Φ holds for all programs of the program composition, until we reach a position where the corresponding continuation satisfies the property Ψ. Note that Φ until Ψ guarantees that the property Ψ holds eventually (strong until).

The on-line introduction of **derived operators** allows a modular and intuitively easy to understand formulation of complex properties. In fact, as can be seen in Figures 4 and 6, our built-in macro facility allows natural language-like specifications. Here we only introduce some convenient derived operators, which are illustrated in Section 4:

$$\begin{array}{lll} \textit{Eventually}: & \textsf{eventually}\ \Phi & =_{df}\ \ \texttt{tt}\ \ \textsf{until}\ \ \Phi \\ \textit{Before}: & \Phi\ \ \textsf{before}\ \ \Psi & =_{df}\ \ (\neg\Psi\ \ \textsf{until}\ \ \Phi)\ \ \textsf{and}\ \ \textsf{eventually}\ \ \Psi \end{array}$$

While the eventually operator requires that Φ will hold at some later point, Φ before Ψ guarantees that Ψ will become valid after Φ has been satisfied but not earlier.

4 The DaCapo Lifecycle

We illustrate the DaCapo lifecycle, sketched in Figure 1, by means of a typical user session.
Starting with the following loosely specified task

> *Display pages 3 and 4 of a latex source in two-page format on the screen*

we are going to synthesize a corresponding custom command from the library of basic commands and applications introduced in Section 2.

[2]Observe that taxonomies are always completed with a top node.
[3]The continuation is simply the program composition starting from the second module
[4]'Postfix' is a straightforward generalization of continuation.

The specification Already the following still natural-language like reformulation of the informal specification is understood by the system (cf. also Figure 4 top right).

```
type latex_source,
     use two_pages,
     use select
         and finally module display
```

Internally, it is transformed into the following SLTL formula

```
type(latex_source)
    and (tt U <two_pages>tt)
    and (tt U <select>tt)
    and (tt U <display>tt)
```

The solution space Given this formula, the system returns the set of four alternative program compositions in graph representation: every path from the start to the success node in the graph of Figure 4 (left) is a possible solution to our synthesis problem. Solutions can be investigated by means of the hypertext browser, which provides information about single modules and types as well as about the type and module taxonomies. The hypertext system offers a card for each node or edge of the graph. As an example, the actual high-level language implementation of the call to the module latex is reported in Figure 4 (right), together with a textual description, which may be complemented by typical examples.

The execution All the proposed solutions are directly executable. In fact, each path in the graph corresponds to a target program in our high level programming language, which is implemented on top of C++. In order to invoke a solution, its corresponding path must be selected by means of the mouse. The system offers the option for execution, as soon as the selection is unambiguous, cf. Figure 5 (front). The chosen solution

(*ltx*, `latex`, *dvi*, `dvips`, *ps*, `psnup -2`, *ps*, `psselect`, *ps*, `ghostview`, *screen*)

which requires an interactive selection of the desired pages as in Figure 5 (upper right), delivers the result shown in Figure 5 (back).

Modifying the query In the *prototyping* phase, the user approaches the desired solution by successively refining or relaxing an initial (usually loose) specification. In fact, the output in Figure 5 reveals that the selection was done on the *double-sided* document[5]. This is unsatisfactory, since it does not reflect the obvious intent, which refers to pages 3 to 4 of the *original* document.

[5]You may recognize that we use *this* paper as source file

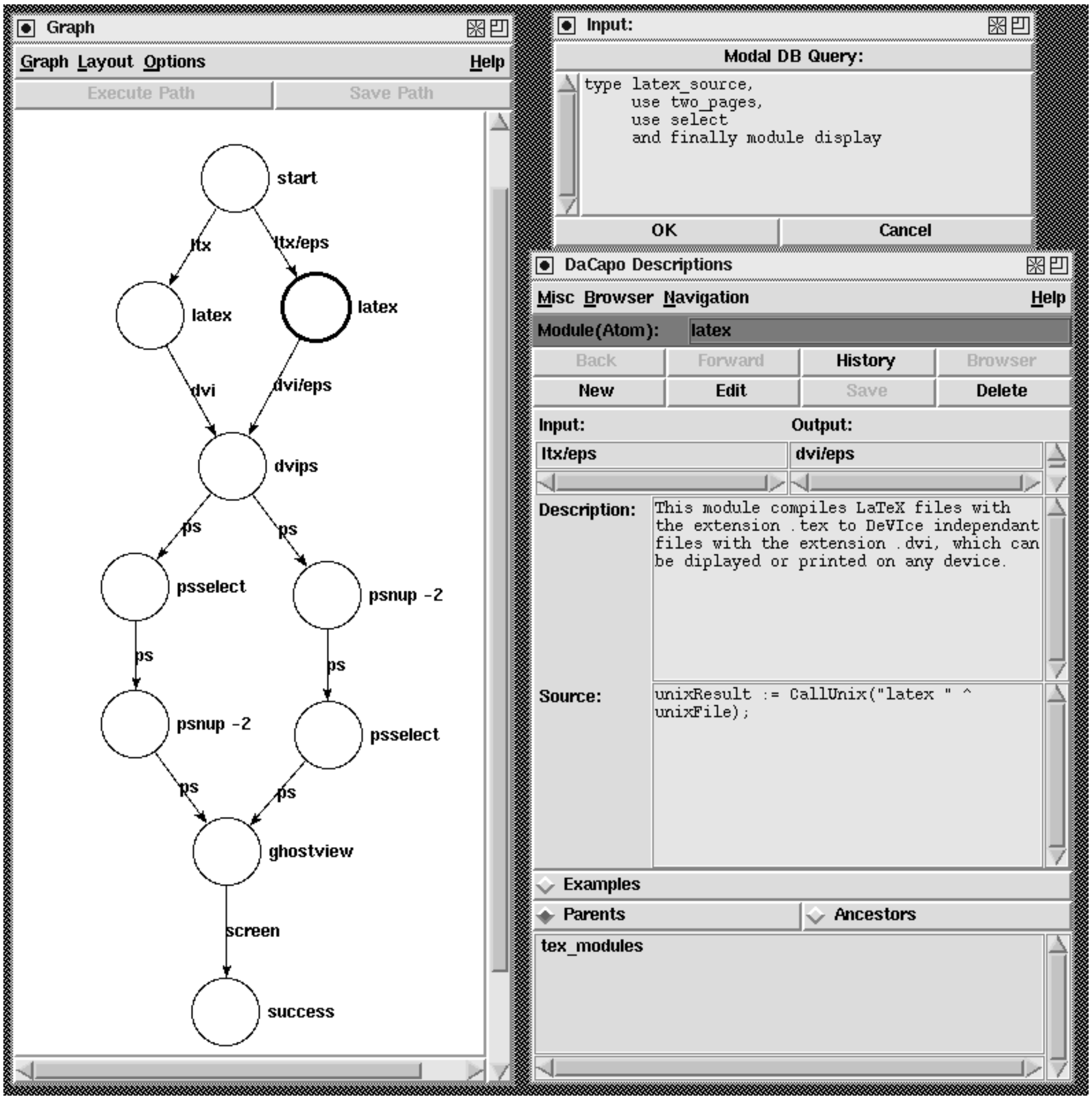

Figure 4: The solution graph

This intuition can be made precise in SLTL by means of ordering constraints requiring the page selection to happen before the two-page formatting:

```
type latex_source,
     module select before module two_pages
     and finally module display
```

as shown in Figure 6 (top right).

Figure 6 (left) shows the solution space corresponding to the modified query, which only contains the correct sequencing (the selection precedes the twopage formatting). Thus the execution of the highlighted path provides the desired result (see Figure 7).

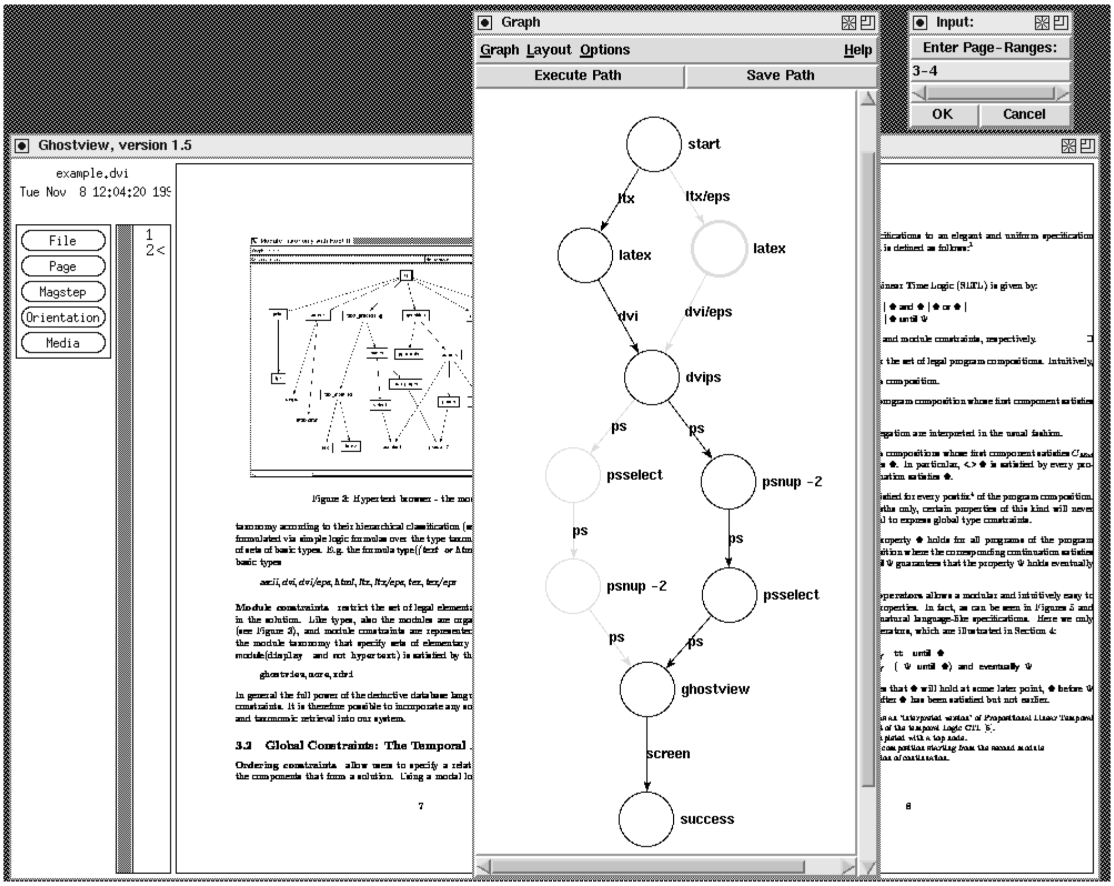

Figure 5: The executed program

The compilation Satisfactory solutions can be made persistent through compilation into a new module that can be saved in the component repository for later reuse. These repository updates are supported by the hypertext system by automatically associating with each new module a new card, which by default contains appropriately precomputed taxonomic criteria and a short textual description as shown in Figure 6 (right). These criteria are generated by the system from the corresponding criteria of its components. Thus the consistency of the taxonomies is automatically maintained. The user only needs to provide a new name, e.g. print_2_latex in this case, then the repository and taxonomy updates follow completely automatically. Of course, the proposed content of the new hypertext card can be also modified interactively.

Figure 7 shows the effect of resubmitting the specification to the updated repository. The execution of the new solution

$$(\texttt{ltx}, \texttt{print\_2\_latex}, \texttt{screen})$$

is identical to the one for its defining path.

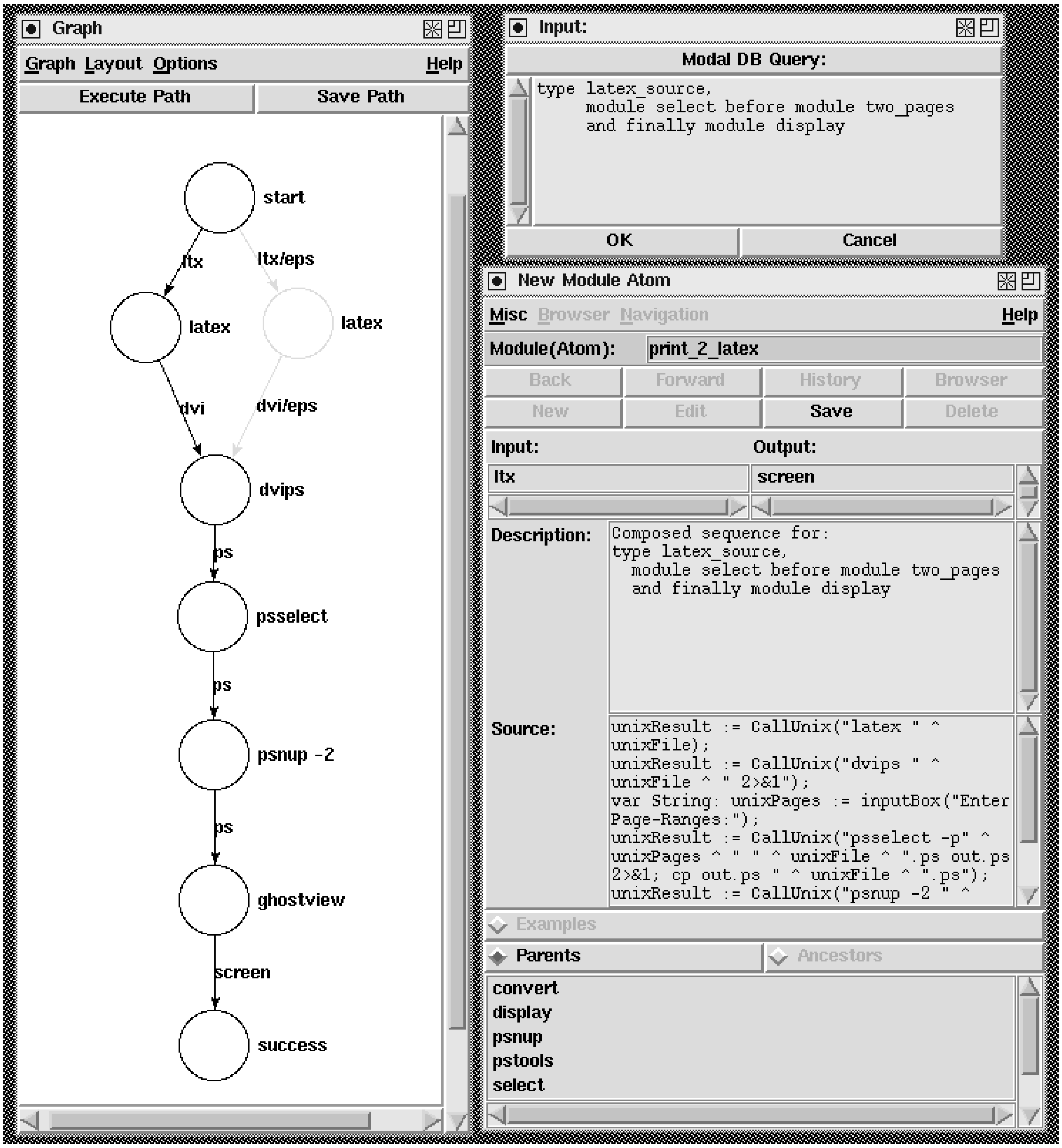

Figure 6: The modified solution graph

5 Comparison with Related Work

Most software library systems described in the literature are **local**, in the sense that they help retrieving single components. E.g.:

- LaSSIE [DBSB91] uses the knowledge representation language KANDOR to store information about a large software project in order to provide semantic retrieval. A taxonomy of classes is induced by a frame-based representation of objects.

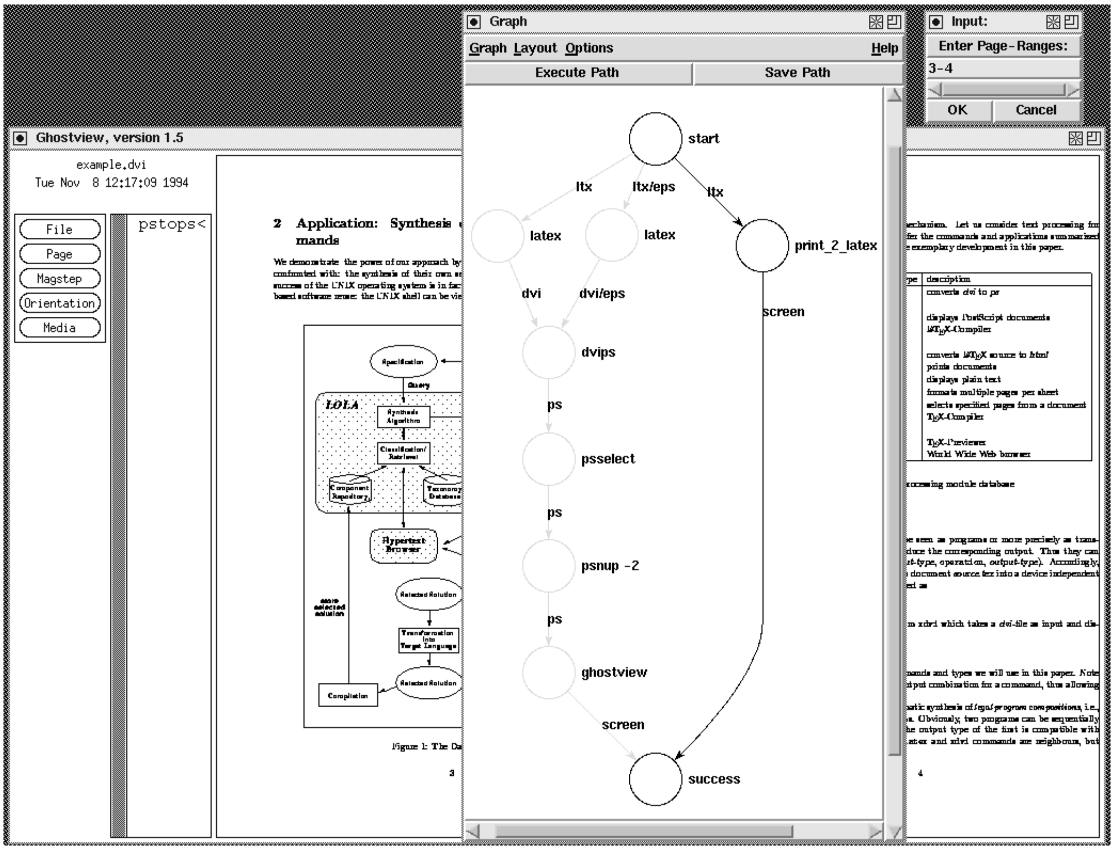

Figure 7: The modified execution

- The software engineering environment NORA [SFGK94] is based on logical inference and supports specification-driven single component retrieval.
- Other systems share a core constituted by the knowledge representation language Telos [MBJK90] and the deductive object-oriented database system ConceptBase [Jark91]:
  - In the ITHACA Environment [FuNP92], an application frame contains application classes with a functional component description in terms of a list of operations and information about the meta-level classes with design knowledge describing the use of components. Component selection is based on similarity in the spirit of Prieto-Diaz' work [Prie91, PrFr87].
  - A logic-based software configuration management system is presented in [JJMG92] using the Telos data model for the representation of configuration information. Given a system interface, the search for an implementation tries to reuse existing subsystems and components if possible. Otherwise, proposals are generated applying the deductive and abductive techniques of ConceptBase.

- An approach based on algebraic specifications and refinement is described in [ChHJ93]. Given an algebraic specification of the desired component in Telos, ConceptBase is used to retrieve software components with matching signatures.

The point of the DaCapo system, however, is its synthesis capability according to **global** constraints. We know only of two other systems with this feature:

- The system reported by Batory and O'Malley [BaMa92] constructs hierarchical software systems from reusable components using grammar rules for high level specification. Their approach strongly relies on structural restrictions of the underlying environment like open architecture software and a strict interface discipline.
- Möhring, Müller and Radermacher, [MöMR94], propose the idea to use scheduling methods to synthesise implementations satisfying precedence constraints. However, they do not provide an appropriate specification language or hints towards its intended expressive power.

The DaCapo system is based on an expressive, well-structured and more intuitive specification language than the first system, and it uses an, in some sense dynamic, scheduling algorithm for synthesis, which varies its search strategy according to the stage of the synthesis.

6 Conclusions

We have presented the DaCapo software library system that supports the *automatic construction of software* from existing components according to a (possibly incomplete) specification. Based on a specification language that uniformly combines taxonomic component specifications, interface conditions, and ordering constraints, our method adds a global view to conventional single component retrieval. Moreover, following a user session, we have illustrated the DaCapo software lifecycle management, which supports the inclusion of a satisfactory new software component into the repository by proposing an appropriately precomputed default taxonomic classification. This guarantees convenient retrieval for later use.

Acknowledgements

We are grateful to Achim Dannecker, Carsten Friedrich, Andreas Holzmann, Dirk Koschützki, Gerald Lüttgen, Falk Schreiber, Matthias Seul, and Barbara Zündorf for their constructive cooperation in the design and implementation of the DaCapo system.

References

[BaMa92] D. Batory, S. O'Malley. The design and implementation of hierachical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, Oct. 1992.

[BuSt92] O. Burkart, B. Steffen. Model checking for context–free processes. In *3rd Int. Conf. on Concurrency Theory (CONCUR '92)*, LNCS Vol. 630, Springer–Verlag, 1992.

[BuSt94] O. Burkart, B. Steffen. Pushdown Processes: Parallel Composition and Model Checking. In *5th Int. Conf. on Concurrency Theory (CONCUR '94)*, Uppsala (Sweden) LNCS Vol. , Springer–Verlag, 1994.

[ChHJ93] P. S. Chen, R. Hennicker,, M. Jarke. On the retrieval of reusable software components. *Advances in Software Reuse - Selected Papers from the 2nd Int. Workshop on Software Reusability, Lucca, Italy*, pp. 99–107. IEEE Computer Society, 1993.

[ClES86] E. Clarke, E. Emerson, A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[ClPS93] R. Cleaveland, J. G. Parrow, B. Steffen. The concurrency workbench: A semantics-based verification tool for finite state systems. *ACM Transactions on Programming Languages and Systems*, Jan. 1993.

[DBSB91] P. Devanbu, R. J. Brachman, P. G. Selfridge, B. W. Ballard. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34(5):35–49, May 1991.

[Emer90] E. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pp. 995–1072. Elsevier, 1990.

[Frei94] B. Freitag. A hypertext-based tool for large scale software reuse. *Proc. 6th International Conference on Advanced Information Systems Engineering, CAiSE'94, Utrecht, The Netherlands, June 1994*, *LNCS* Vol. 811, pp. 283–296, Berlin, 1994. Springer-Verlag.

[FrKe94] B. Freitag, J. Kempe. Multidatabase access with the *LOLA* system. Technical Report TUM-I9416, Technische Universität München, Munich, Germany, 1994.

[FrMS94] B. Freitag, T. Margaria, B. Steffen. A pragmatic approach to software synthesis. In *Proc. ACM SIGPLAN POPL'94 Post-Conference Workshop on Interface Definition Languages*, Portland, Oregon, Jan. 1994. (ACM SIGPLAN Notices, 1994).

[FrSS91] B. Freitag, H. Schütz, G. Specht. *LOLA* - a logic language for deductive databases and its implementation. *Proc. 2nd Intl. Symp. on Database Systems for Advanced Applications (DASFAA '91)*, pp. 216 – 225, Tokyo, 1991.

[FuNP92] M. Fugini, O. Nierstrasz, B. Pernici. Application development through reuse: the ITHACA tools environment. *ACM SIGOIS Bulletin*, 13(2):38–47, Aug. 1992.

[Jark91] M. Jarke. ConceptBase V3.0 User Manual. Technical Report MIP-9106, Universität Passau, 1991.

[JJMG92] M. Jarke, M. Jeusfeld, A. Miethsam, M. Gocek. Towards a logic-based reconstruction of software configuration management. In *Proc. of the seventh Knowledge-Based Software Engineering Conference, McLean, Virginia*, pp. 132–141. IEEE Computer Society Press, 1992.

[KaRT89] S. Katz, C. Richter, K. The. PARIS: A system for reusing partially interpreted schemas. In *Software Reusability: Volume I — Concepts and Models*, Frontier Series, chapt. 10, pp. 257–273. ACM Press, New York, 1989.

[KnRS92] J. Knoop, O. Rüthing, B. Steffen. Lazy code motion. *Proc. PLDI '92*, *ACM SIGPLAN Notices*, Vol.27, pp. 224 - 234, San Francisco, June 1992.

[KnRS94a] J. Knoop, O. Rüthing, B. Steffen. Partial Dead Code Elimination, *Proc. SIGPLAN PLDI Conf.'94*, *ACM SIGPLAN Notices*, Orlando (FL), June 1994.

[KnRS94b] J. Knoop, O. Rüthing, B. Steffen. *Optimal Code Motion: Theory and Practice. ACM Transactions on Programming Languages and Systems*, Vol. 16, N.4, July 1994, pp.1117-1155.

[MiMM94] A. Mili, R. Mili, R. Mittermeier. Storing and retrieving software components: A refinement based system. *Proc. 16th International Conference on Software Engineering, Sorrento, Italy.* IEEE Computer Society, 1994.

[MöMR94] R. H. Möhring, R. Müller, F. J. Radermacher. Advanced DSS for scheduling: Software engineering aspects and the role of eigenmodels. In *Proc. 27th Annual Hawaii Int. Conf. on System Sciences, Maui, Hawaii*, 1994.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis. Telos – representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, Oct. 1990.

[Oust94] John K. Ousterhout. TCL and the TK Toolkit. Addison–Wesley, April 1994.

[Prie91] R. Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5), May 1991.

[PrFr87] R. Prieto-Diaz, P. Freeman. Classifying software for reusibility. *IEEE Software*, 4(1):6–16, Jan. 1987.

[SiCl85] A. P. Sistla, E. M. Clarke. The complexity of the propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, July 1985.

[SFGK94] G. Snelting, B. Fischer, F. J. Grosch, M. Kievernagel, A. Zeller. Die inferenzbasierte Softwareentwicklungsumgebung NORA. *Informatik Forschung und Entwicklung*, 9(3), Aug. 1994. (In German).

[Stef93] B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, N. 21, pp. 115-139, 1993. (a shorter version appeared in the*Proc. TACS '91*, Sendai, Japan, Sept. 1991, *LNCS* N. 526, pp. 346–364, Springer–Verlag.)