Construction of application generators using Eli

Uwe Kastens

Fachbereich Mathematik/Informatik
Universität-GH-Paderborn
Warburger Str. 100
D - 33098 Paderborn

Bericht tr-ri-94-143
Reihe Informatik
März 1994

Construction of Application Generators Using Eli

Uwe Kastens, University of Paderborn, FRG

Abstract

Application generators produce special purpose programs from very high-level descriptions. That principle is applicable in application domains where variants of programs are developed that solve different instances of a certain application problem, e. g. data base report generators. Application generators are a powerful means for reuse of software design. The structure of application generators is similar to that of programming language compilers: They translate from a domain specific description language into a programming language. In this paper we show that the Eli system, an integrated toolset for language implementation, is well suited for construction of application generators.

1 Introduction

Application generators are software systems that generate programs for different instances of a problem in a specialized application domain. Typical examples are programs that produce reports of data extracted from a data base. Such a program can be systematically constructed from information describing the data base and the desired layout. Many different variants of such programs may be needed to produce different reports. An application generator for this problem domain generates such a program from a specification that describes which data have to be retrieved and which layout has to be produced.

An application generator encorporates the knowledge how to construct such programs. Users of the application generator can produce programs in that application domain by describing *what* they shall do without the need to understand *how* they are built.

The input for an application generator can be considered as a very high level specification within a restricted problem domain. It is formulated in a dedicated specification language. An application generator is a translator for that language. Hence techniques for language implementation have to be applied for the construction for application generators.

In this paper we show that Eli [2,9], an integrated toolset for language implementation, is well suited for constructing application generators. Eli incorporates state-of-the-art know-how in the field of compiler construction. It enables users who need not be compiler specialists to produce language implementations of high quality.

This is most important for application generators, since they are usually constructed by people who are specialists in an application domain rather than in

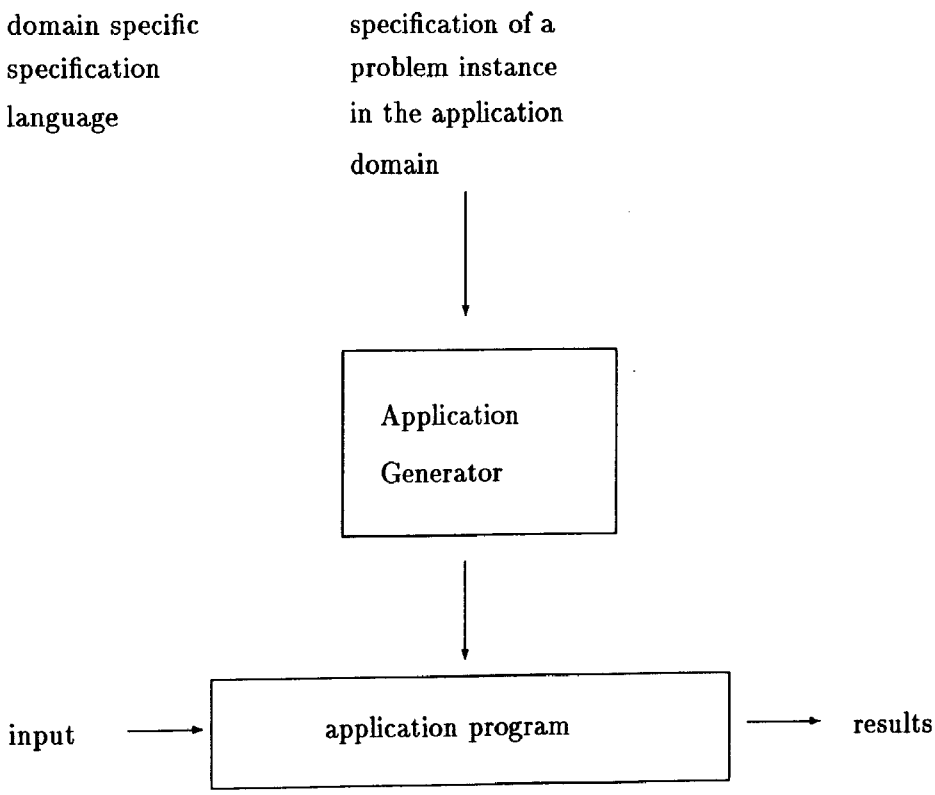

Figure 1: Application Generators

language implementation. Eli itself can be considered as an application generator for the domain of language implementation. Eli has been successfully applied in many industrial and academic projects, and in many application domains.

The term application generator was initially coined in [3] for the problem domain of data base tools. Cleaveland [1] generalizes the principle of application generators, points out their relationship to compilers, and describes tools to construct them. In [7] application generators are discussed as one of several principles for software reuse.

In the rest of the paper we summarize the principle of application generators and present strategic aspects that make Eli suitable for application generator construction. We demonstrate these strategies using data base report generators as a running example throughout this paper.

2 Application Generators

An application generator produces variants of programs that solve instances of problems in a certain restricted domain. Each problem instance is described in terms of a dedicated specification language, the input language for the application generator, Figure 1.

Let us assume for example that we want to produce information extracted from a data base containing publication references. Such a data base report may look like

```
Papers of Kastens since 1991:

1991    An Abstract Data Type for Name Analysis
1991    Attribute Grammars as a Specification Method
1991    An Attribute Grammar System in a Compiler Construction Environment
1991    Implementation of Visit-Oriented Attribute Evaluators
1992    Modularity and Reusabiltiy in Attribute Grammars
1993    Executable Specifications for Language Implementation

Kastens published 6 papers.
```

It may be useful to get such reports for different authors and time spans. Hence, we need a program to produce the report which takes the author and the year as input. Furthermore we may want to have several such programs that produce reports containing different information presented in a certain layouts. According to the application generator principle we decribe the desired report in a specification language designed exactly for that problem class. The application generator translates such a description into an application program.

The description for the above example may be

```
string name = InString ("Which author?");
int since   = InInt ("Since which year?");
int cnt     = 0;

"Papers of ", name, " since ", since, ":\n\n";
[
SELECT SubString (name, Author) && Year >= since;
cnt = cnt + 1;

Year, "\t", Title, "\n";

]
"\n", name, " published ", cnt, " papers.\n\n\n";
```

It describes which records are to be selected from the data base, the fields to be extracted, and the layout of the desired report. The generated program consists of operations that access the data base and operations that produce the formatted output. Those operations are selected and parameterized according to the description and are systematically composed according to technical requirements for data base use and printing facilities.

The same principle can be used in other problem domains: An application generator for simulation of mechanical systems may take descriptions of system structures and of feedback equations for their components as input. An instance may be the description of a car suspension. The generated simulator program can be run to validate that car suspension under different constraints.

The principle of application generators is usefully applied in problem domains that have certain appropriate characteristics. Krueger [7] describes them as follows:

a) If many similar software systems are written.

Structuring	Lexical analysis	Scanning Conversion
	Syntactic analysis	Parsing Tree construction
Translation	Semantic analysis	Name analysis Type analysis
	Transformation	Data mapping Action mapping
Encoding	Code generation	Execution-order determination Register allocation Instruction selection
	Assembly	Instruction encoding Internal address resolution External address resolution

Figure 2: Compilation Subproblèms

b) If one software system is modified or rewritten many times during its lifetime.

c) If many prototypes of a system are necessary to converge to a usable product.

The example of the data base report programs fits to each of these characteristics: Many different forms may be needed (a). The report programs have to be modified when the data base schema changes (b). Different layouts of a form may be considered until the final version is chosen (c).

Different people may be involved in the construction and use of an application generator and its products:

a) the constructor of the application generator,

b) the user of the application generator,

c) the user of the generated application program.

The principle of application generators allows that these people have different knowledge in different areas and act in roles suitable for their knowledge. In our example of the data base report generator a specialist for a bibliographic data base develops the application generator (a). He knows how to access the data base how to print extracted information, and how to construct programs to solve these tasks.

He also designs the specification language and implements its analysis and its translation into target programs. Solutions of these tasks clearly require language implementation techniques. In order to avoid the application generator constructor to be specialist in both areas he needs tools that incorporate language implementation knowledge.

A librarian (b) uses the application generator to produce a set of report programs. He knows which information is in the data base and what the frequently asked questions of library users are. Finally the library users (c) call these programs to retrieve information. They only need to know which kind of reports are provided and how the corresponding programs are called.

3 Problem Decomposition

Implementation of application generators can be considered as a language implementation task: Specifications of problem instances written in a domain specific language are translated into application programs. Well established knowledge and effective tools for translator construction can be applied to solve that task. The domain specific aspects of the solution then concentrate on the design of the specification language and on the components of which application programs are composed of.

Many years of research and experience in the field of compiler construction led to a generally accepted model for decomposition of compilation tasks, as shown in Figure 2 taken from [2]. Lexical analysis transforms the input program representation from a stream of characters into a stream of tokens, e. g. identifiers, numbers, operators, and keywords. It determines the role of each token within the program structure, and stores and encodes values associated with tokens. Syntactic analysis determines the program structure by parsing the token stream and represents the result in form of a tree. Each node together with its subtree is an abstraction of an occurrence of a language construct in the input program, e. g. a declaration, statement, or expression.

All tasks of the subsequent translation phase are expressed by computations associated to nodes of that abstract program tree. The name analysis tasks maps occurrences of identifiers to internal representations of named objects, e. g. variables, parameters, or record fields, according to the scope rules of the language. Type analysis associates the type property to objects and expression nodes and validates typing restrictions of the language.

The transformation task maps the data objects of the program onto the storage of an execution model, and transforms statements and operations into an instruction sequence for that model. For programming language compilers that model is usually defined by an intermediate language. It serves as an interface to separate the encoding phase from the frontend of the compiler. That final phase translates the intermediate program representation into target machine code.

This decomposition model is applicable for the implementation of domain specific languages as well as for programming languages. Let us consider our example of data base report generators.

Structuring. The first step in the design of a description language defines the structural properties of the information to be described. In our case the overall structure should be chosen according to the report structures, being composed of three sections for the header, the record selection part, and the summary. Each section is a sequence of statements that may have one of several forms, e. g. an output statement is described by a sequences of string literals and expression for text to be inserted:

```
"Papers of ", name, " since ", since, ":\n\n";
```

The complete structuring task refines such structural components down to the level of tokens, and describes the token notation.

Name Analysis. Our description language has two kinds of named objects: those introduced by definitions and the record field names. Each occurrence of a certain name refers to the same object. We require that the definition of a name preceeds its applications, and that no name is multiply defined. Record field names are predefined by the data base interface.

Type Analysis. Our language needs values of three types: strings and integral numbers for being printed, and boolean values that determine selection. Opera-

tors over these types and calls of predefined functions (e. g. input functions like `InInt`) should be type checked. Definitions associate the type property to the defined object. We also have to distinguish three kinds of objects: defined variables, predefined record fields, and predefined functions. This distinction is similar to the type property, hence it belongs to this subtask.

Transformation. The constructs of the description language are mapped to components of the application program, a C program in our case. As an example for this transformation we consider the header line of our report description:

```
"Papers of ", name, " since ", since, ":\n\n";
```

It is to be transformed into a C `printf` statement:

```
printf (%s%s%s%d%s",
"\n\nPapers of ", name, " since ", since, ":\n\n");
```

The format string is derived from the types of the expression sequence. The remaining arguments result from the translation of the expression sequence, which is an identical mapping in this simple example.

The whole transformation task requires the design of such mappings for all relevant language constructs, and their composition. The composition is trivial where the structure of the description and that of the target program coincide, as in the example above. The transformation of definitions are an example for restructuring: Definitions may occur anywhere in a report description; their translations have to be collected at the beginning of the C program. The transformation is also influenced by properties that result from the analysis task, e. g. the format string above. Similarly the kind of a named object determines whether its occurrences are translated into a variable name or a call of a data base access function in case of record field names.

The transformation task maps directly to the target program. Hence, the encoding task of the decomposition model for programming language compilers (Figure 2) are not needed her. This situation is typical for application generators.

4 Problem Solution Methods

In this section we demonstrate how Eli supports application generator construction. We emphasize the consequences of the problem decomposition model and the solution methods typically applied.

Eli comprises a huge number of components each dedicated for the solution of a subtask in the decomposition model discussed in the previous section. There are several generating tools including generators for scanners, parsers, grammar mapping, attribute evaluators, operator identification, definition tables, and output transformers. These tools are driven by dedicated specification languages, see Figure 3.

A library of C modules provides implementations of basic tasks that are common in language implementation, e. g. input reading, storing and encoding identifiers and literals, error message output, consistent renaming of identifiers. Their implementations are generally applicable, validated and efficient according to the state-of-the-art in compiler construction.

A library of specification modules contains complete solutions of particular subproblem instances which occur often in language implementation, e. g. different

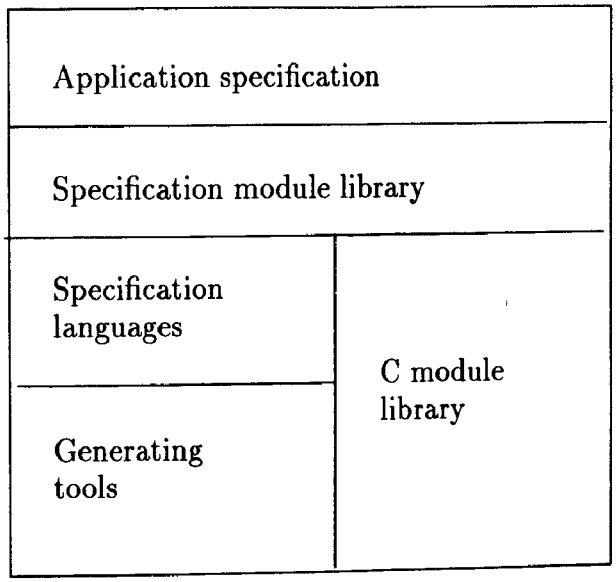

Figure 3: Problem solution with Eli

scope rules for consistent renaming of identifiers. Such specification modules comprise input fragments for generators and applications of basic library functions. They can be easily reused for the specification of different application tasks.

The components of Eli are smoothly integrated on the base of the task decomposition model. Hence, solutions of related subtasks fit together, usually without any consideration of the user. In some cases Eli even deduces the existence of a certain task and supplies its solution. An example is given below for the structuring subtask.

Eli users may chose for each task the best suited solution strategy from any layer of components in Figure 3. From a general point of view the solution of a particular problem P can be described by one of three methods [8]:

1. Describe the properties of the problem P. Such a description is expressed in a specification language for one of Eli's generating tools.
2. Identify P with the description of a problem Q that has a known solution. This is the principle of Eli's library of specification modules.
3. Describe a solution of P. This strategy is completely operational. The implementation of a subproblem solution is provided by the user. It can be considered as an escape for subtasks not foreseen in Eli. It is be applied if certain domain specific algorithms have to be integrated into the application generator, e. g. the data base access functions in the case of our example. Eli then supports integration and interfacing with solutions of related subtasks.

For complex subtasks several of the three solution strategies may be combined. The following examples demonstrate how the methods (1) and (2) are applied to solve subtasks for the report generator as described in the previous section.

Structuring. The central aspect of this task is the structure of report descriptions. It is specified using a notation for context-free grammars (method 1).

```
Report:     Source.
Source:     Section '[' Section ']' Section.
Section:    Statements.
```

```
Statements:     Statements Statement / .

Statement:      OutputItems ';'.
OutputItems:    OutputItems ',' OutputItem / OutputItem.
OutputItem:     Expr.

Statement:      'SELECT' Expr ';'.

Statement:      Type DefIdent  '=' Expr ';'.
Type:           'int' / 'string' / 'bool'.
DefIdent:       Ident.

Statement:      UseIdent '=' Expr ';'.
```

It describes the decomposition into three sections, each consisting of a statement sequence. A statement may be a sequence of output expressions, a selection expression, a variable definition, or an assignment. The restriction that selections may only occur in the central section is deferred to the semantic analysis task where it can be easily checked, instead of a more complicated description here.

We do not further refine expressions. Instead we instruct Eli to add a reusable module from the library (method 2). It describes operators for the three types, literals, identifiers, and function calls in C-like notation. Eli allows to combine several fragments of each specification class: here one is explicitly specified and one is taken from a library.

The structure description given above covers the whole structuring task. It is obvious that the parsing subtask can be solved by a parser generator given the context-free grammar. The existence of the tree construction task and its solution are deduced automatically by Eli: The description of the transformation task (see example below) states which tree nodes are needed. An Eli tool determines how they are to be generated by the parser.

Some properties of the lexical analysis task are already described in the context-free grammar, e. g. the notation of tokens like `SELECT`, `;`, etc. The descriptions of identifiers, string literals, and numbers come from the library's expression module, e. g.

```
Ident: C_IDENTIFIER
```

This again refers to a precoined solution for scanning, storing, and encoding of identifiers written as in C. If we did not use that expression module, we could have added that lexical specification directly, or chosen a different language style, or describe our own identifier notation.

Name Analysis. We use this subtask to demonstrate that an individual solution is easily achieved by a selection of combinable library modules (method 2).

All problem instances of the name analysis task can be reduced to a small set of basic concepts [5]: Occurrences of names denote objects (e. g. variables). Within each range of the input text all occurrences of one name denote the same object. The binding between a name and its object is established by a definition of the name for the smallest enclosing range. A use of a name yields the object bound to it in the smallest enclosing range that has such a binding, if any.

Eli's name analysis library provides modules which implement these concepts, **Range IdDef**, **IdUse**, and **Root** (the latter for a structure that encloses all ranges).

[6] For our task we chose the Chain module that establishes the bindings from the definition upto the end of the range. These concepts are simply associated to symbols of our grammar indicating that they play the corresponding role:

```
SYMBOL Report   INHERITS RootChain END;
SYMBOL Source   INHERITS RangeChain, RangeUnique END;
SYMBOL DefIdent INHERITS IdDefChain, IdDefUnique, IdentSym END;
SYMBOL UseIdent INHERITS IdUseChain, NoKeyMsg END;
SYMBOL FctIdent INHERITS IdUseChain, NoKeyMsg END;

ATTR Sym:       int;
SYMBOL IdentSym COMPUTE SYNT.Sym = CONSTITUENT Ident.Sym; END;
```

The above is written in the specification language LIDO for the attribute evaluator generator LIGA [4]. The same technique is used for the additional requirements of our description language: no multiply definitions (module `Unique`), and each used name is defined (module `NoKeyMsg`).

Those translation tasks which can not completely solved by library modules (e. g. type analysis) are specified in the language LIDO: Computations, written as C function calls, are associated to tree nodes. Dependencies between computations in different tree contexts are stated as pre- and postconditions. Such a specification abstracts from the evaluation order during the tree walk which is automatically generated.

Transformation. This task is solved by the combination of two descriptions: One specifies a set of target text patterns. Eli's PTG tool generates functions from it which compose and output the text. The other describes which text patterns are to be produced for different tree nodes. It is specified in terms of calls of these functions and their dependencies, and is written in LIDO. This technique is typical for language processors which translate into high level programs, like application generators.

Here only the translation of the output statements of our report description language is shown. Each output statement is translated into a C `printf` statement as described in the previous section.

The pattern for the `printf` statement

```
PrintStmt:
  "printf (\"" $1 /*formats*/ "\", " $2 /*args*/ ");\n"
```

consists of three string literals and two variables ($1, $2) where the format string and the expressions to be printed are inserted. They are supplied as arguments of the calls of the generated function `PTGPrintStmt`.

The pattern is applied at the subtree root of each output statement in the description:

```
CHAIN CFormat, CPrintArgs:      PTGNode;

RULE: Statement ::= OutputItem ';' COMPUTE
  CHAINSTART HEAD.CFormat = PTGNULL;
  CHAINSTART HEAD.CPrintArgs = PTGNULL;
  Statement.CStmt = PTGSeq (Statement.CStmt,
  PTGPrintStmt (TAIL.CFormat, TAIL.CPrintArgs));
END;
```

The two arguments are both obtained from the sequence of `OutputItems` in its subtree. Each `OutputItem` contributes one component to each argument. They are composed in left-to-right order specified using LIDO's `CHAIN`-construct: The `CHAIN`s `CFormat` and `CPrintArgs` are started in this tree context. Their results obtained by `TAIL.CFormat` and `TAIL.CPrintArgs` fill the `PrintStmt` pattern, which then is appended to an outer `CHAIN` that collects the translation of all statements in a section.

The following LIDO fragment describes the contribution of each `OutputItem` to the `CHAIN` that composes the format string:

```
RULE: OutputItem ::= Expr COMPUTE
OutputItem.CFormat = PTGSeq (OutputItem.CFormat,
IF (EQ (Expr.Type, Tint),  PTGIntFormat (),
IF (EQ (Expr.Type, Tbool), PTGBoolFormat (),
   PTGStringFormat ())));
END;
```

It selects one of three formats depending on the expression type which is determined by the type analysis task. The patterns used for the format items are trivial:

```
StringFormat:   %s"
IntFormat:      %d"
BoolFormat:     %c"
```

The translation of expressions into C code using PTG specifications are obtained from the expression module in the library. Hence, it need not be described explicitly.

5 Conclusion

In this paper we have shown that the principle of application generators may effectively reduce software development efforts in certain application domains. As the development of application generators can be considered as a language implementation problem a tool set like Eli is well suited for their construction. Eli is designed such that it can be used without specialized knowledge in language implementation techniques. Hence, constructors of application generators need not be specialists in both areas, their application domain and language implementation.

Using data base report generators as an example for application generator we demonstrated Eli's central paradigms: a straight-forward problem decomposition model supported by cooperating tools, high level specification languages dedicated for different tasks, and large libraries that provide reusable and combinable solutions for common tasks. These facilities allow for effective construction of application generators from descriptions on adequate levels of abstraction. (The complete specification for the presented example is available with the Eli system.)

Eli has proven to be a useful toolset in many real life projects in different domains, e. g mechanical systems simulation, data base tools, user interface tools, software configuration, signal processing, graphical music scores, and of course compiler construction. Eli is developed in cooperation of the group of W. M. Waite, University of Colorado, Boulder and the group of the author. It is available via anonymous `ftp` from both institutions.

Acknowledgements

This work was partially supported by the government of Nordrhein-Westfalen through the SofTec-Cooperation, and by the US Army Research Office under grant DAAL03-92-G-0158.

6 References

1. Cleaveland, J. C., "Building Application Generators," *IEEE Software* 5 (July 1988), 25–33.
2. Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M. & Waite, W. M., "Eli: A Complete, Flexible Compiler Construction System," *Communications of the ACM* 35 (February 1992), 121–131.
3. Horowitz, E., Kemper, A. & Narasimhan, B., "A Survey of Application Generators," *IEEE Software* 2 (Jan 1985), 40–54.
4. Kastens, U., "LIDO - Short Reference," University of Paderborn, FRG, Documentation of the LIGA System, 1992.
5. Kastens, U. & Waite, W. M., "An Abstract Data Type for Name Analysis," *Acta Informatica* 28 (1991), 539–558.
6. Kastens, U. & Waite, W. M., "Modularity and Reusability in Attribute Grammars," Universität-GH Paderborn, Reihe Informatik, July 1992.
7. Krueger, C. W., "Software Reuse," *ACM Computing Surveys* 24 (June 1992), 131–183.
8. Waite, W. M., "A Complete Specification of a Simple Compiler," Department of Computer Science, University of Colorado, Boulder, CU-CS-638-93, Jan 1993.
9. Waite, W. M., Heuring, V. P. & Kastens, U., "Configuration Control in Compiler Construction," in *International Workshop on Software Version and Configuration Control '88*, Teubner, Stuttgart, 1988.

Reihe I N F O R M A T I K

97. Kutyłowski, M./Wanka, R.: Periodic sorting on two-dimensional meshes. Januar 1992

98. Wanke, E.: Bounded tree-width and LOGCFL. Februar 1992

99. Höfting, F./Wanke, E.: Minimum cost paths in periodic graphs. März 1992

100. Wanke, E.: The complexity of connectivity problems on context-free graph languages. Mai 1992

101. Lettmann, T./Schmitgen, S.: Auswahl- und Positionierungsprobleme beim Konfigurieren und ihre aussagenlogische Beschreibung. Juni 1992

102. Kastens, U./Waite, W.M.: Modularity and reusability in attribute grammars. Juli 1992

103. Kastens, U./Pfahler, P. (eds.): International Workshop on Compiler Construction CC'92. Extended Abstracts. Oktober 1992

104. Lengauer, T./Heistermann, J.: Hierarchical compaction and the solution of parameterized path problems. Oktober 1992

105. Wichmann, F./Pfahler, P.: Compilation for fine-grained parallelism: A code generator for the Intel i860. Oktober 1992

106. Kleine Büning, H./Karpinski, M./Flögel, A.: Resolution for quantified Boolean formulas. November 1992

107. Flögel, A./Kleine Büning, H./Lettmann, T.: On the restricted equivalence for subclasses of propositional logic. November 1992

110. Buro, M./Kleine Büning, H.: Report on a SAT competition. November 1992

111. Reski, T.: Parallele Simulation des Backpropagation Lernalgorithmus. November 1992

112. Klasing, R./Lüling, R./Monien, B.: Compressing cube-connected cycles and butterfly networks. Februar 1993

113. Klasing, R./Monien, B./Peine, R./Stöhr, E.A.: Broadcasting in butterfly and De-Bruijn networks. Februar 1993

114. Lüling, R./Monien, B.: Load balancing for distributed branch & bound algorithms. Februar 1993

115. Feldmann, R./Hromkovič, J./Madhavapeddy, S./Monien, B./Mysliwietz, P.: Optimal algorithms for dissemination of information in generalized communication modes. Februar 1993

116. Feldmann, R./Mysliwietz, P.: The shuffle exchange network has a Hamiltonian path. Februar 1993

117. Diekmann, R./Lüling, R./Simon, J.: Problem independent — Distributed simulated annealing and its applications. Februar 1993

118. Hromkovič, J./Procházka, J.: On the optimality of Kung's convolution algorithm in some classes of systolic algorithms. März 1993

119. Pfahler, P.: Karel the Robot. An exercise in implementing language processors with Eli. März 1993

120. Reinefeld, A./Marsland, T.A.: Enhanced iterative-deepening search. März 1993

121. Ladkin, P.B./Reinefeld, A.: A symbolic approach to interval constraint problems. März 1993

122. Funke, R./Lüling, R./Monien, B./Lücking, F./Blanke–Bohne, H.: An optimized reconfigurable architecture for transputer networks. März 1993

123. Kröger, B./Lüling, R./Monien, B./Vornberger, O.: An improved algorithm to detect communication deadlocks in distributed systems. März 1993

124. Monien, B./Feldmann, R./Klasing, R./Lüling, R.: Parallel architectures: Design and efficient use. März 1993

125. Menzel, K.: Report on distributed virtual reality. April 1993

126. Menzel, K./Ohlemeyer, M.: Walking-through animation in three-dimensional scenes on massively parallel systems. April 1993

127. Lürwer-Brüggemeier, K./Meyer auf der Heide, F.: Capabilities and complexity of computations with integer division. Juni 1993

128. Diekmann, R./Menzel, K./Stangenberg, F.: How to implement distributed algorithms efficiently. Juni 1993

129. Hromković, J./Klasing, R./Stöhr, E.A.: Dissemination of information in vertex-disjoint paths mode. Part 1: General bounds and gossiping in hypercube-like networks. August 1993

130. Hromković, J./Klasing, R./Stöhr, E.A./Wagener, H.: Dissemination of information in vertex-disjoint paths mode. Part 2: Gossiping in d-dimensional grids and planar graphs. September 1993

131. Pardubská, D.: On the power of communication structure for distributive generation of languages. September 1993

132. Reinefeld, A.: A minimax algorithm faster than alpha-beta. September 1993

133. Pardubská, D.: Communication complexity hierarchies for distributive generation of languages. September 1993

134. Karp, R.M./Luby, M./Meyer auf der Heide, F.: Efficient PRAM simulation on a distributed memory machine. September 1993

135. Kleine Büning, H./Lettmann, T.: Search space and average proof length of resolution. November 1993

136. Hromković, J./Klasing, R./Pardubská, D./Unger, W./Wagener, H.: The complexity of systolic dissemination of information in interconnection networks. Dezember 1993

137. Hromković, J./Kari, J./Kari, L./Pardubská, D.: Two lower bounds on distributive generation of languages. Januar 1994

138. Feldmann, R./Mysliwietz, P./Monien, B.: Game tree search on a massively parallel system. Januar 1994

139. Feldmann, R./Mysliwietz, P./Monien, B.: Experiments with a fully distributed chess program. Januar 1994

140. Nagel, C.: Selection of test points during high-level synthesis. Februar 1994

141. Buro, M.: The 1st International Paderborn Computer-Othello Tournament. Februar 1994

142. Dunker, U./Flögel, A./Kleine Büning, H./Lehmann, J./Lettmann, T.: ILFA - A project in experimental logic computation. März 1994

143. Kastens, U.: Construction of application generators using Eli. März 1994

Reihe F O R S C H E R G R U P P E

1. Höfting, F./Wanke, E.: Minimum cost paths in periodic graphs. März 1992
2. Wanke, E.: Paths and cycles in 3-dimensional conditional non-bounded dependence graphs. Dezember 1992
3. Meyer auf der Heide, F./Oesterdiekhoff, B./Wanka, R.: Strongly adaptive token distribution. Februar 1993
4. Monien, B./Lüling, R./Langhammer, F.: A realizable efficient parallel architecture. März 1993
5. Wachsmann, A./Wichmann, F.: OCCAM-light — A multiparadigm programming language for transputer networks. April 1993
6. Meyer auf der Heide, F.: Hashing strategies for simulating shared memory on distributed memory machines. Juni 1993
7. Kik, M./Kutyłowski, M./Stachowiak, G.: Periodic constant depth sorting networks. Oktober 1993