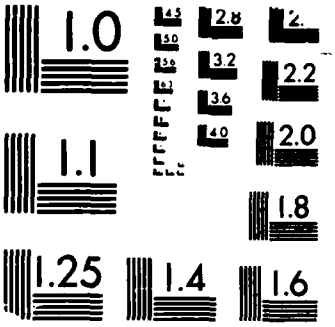
AD-A193 916 CONFIGURATION CONTROL IN COMPILER CONSTRUCTION(U) 1/1
COLORADO UNIV AT BOULDER SOFTWARE ENGINEERING GROUP
W M WAITE ET AL. SEP 87 N00014-86-K-0204

UNCLASSIFIED F/G 21/5 NL



MICROCOPY RESOLUTION TEST CHART
(BUREAU OF STANDARDS 1963-A

AD-A193 916

DTIC FILE COPY

(4)

CONFIGURATION CONTROL IN COMPILER CONSTRUCTION

W. M. Waite
V. P. Heuring
U. Kastens †

Department of Electrical and Computer Engineering
University of Colorado
Boulder, Colorado

DTIC ELECTE MAR 1 0 1988 S D C&R

DISTRIBUTION STATEMENT A
Approved for public release
- Distribution Unlimited

88 3 1 16

Software Engineering Group Report No. 87-1-4

CONFIGURATION CONTROL IN COMPILER CONSTRUCTION

W. M. Waite
V. P. Heuring
U. Kastens †

Department of Electrical and Computer Engineering
University of Colorado
Boulder, Colorado 80309-0425
USA

MAR 1 0 1988

compiler@boulder.csnet
compiler.boulder@csnet-relay.arpa
{seismo!hao,decvax!sunybcs,ucbvax!nbires}!boulder!compiler

September 1987

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

† Present address: Universität-GH Paderborn, 4790 Paderborn, West Germany

Configuration Control in Compiler Construction

W. M. Waite
V. P. Heuring
U. Kastens †

Department of Electrical and Computer Engineering
University of Colorado
Boulder, Colorado 80309-0425
USA

ABSTRACT: A compiler is made up of a large number of individual components. Some of these components are generated from formal specifications, some result from hand adaptation of general algorithms, and some are standard modules from a library. Configuration control is a serious problem in compiler construction: How do we subdivide the compilation task into components, solve them individually, and then re-integrate the solutions into a consistent product. We have successfully used the Odin object manager to solve this problem for a particular compiler architecture. The resulting system illustrates many of the complexities of configuration control in an environment of compiler construction tools.

[Keywords: implementation]

DTIC COPY INSPECTED

Accesion For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By per HP	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

† Present address: Universität-GH Paderborn, 4790 Paderborn, West Germany

1. Introduction

Tool support is currently available for the majority of compiler subtasks. Existing tools were developed individually, and often little thought was given to how each tool might interact with tools supporting other subtasks. The overall result is that the same information must appear in specifications for several different tools, and the modules or code fragments produced by distinct tools do not work together smoothly. Moreover, the code generated by many tools has significantly poorer performance than the equivalent code produced by hand.

Despite these disadvantages, generation of a compiler from formal specifications is important because those specifications can be machine-checked for consistency. This means that, although a tool-generated compiler may not behave as desired, it will not crash. In effect, the product is a correct compiler for a language or machine that differs from the one intended. Such errors in specification are generally easier to find and correct than errors due to inconsistent data structures or incorrect algorithms.

The disadvantages of compiler generation fall into two broad classes:

- Inadequacy of individual tools.
- Complexity of the generation process.

Problems in both classes must be solved if generation is to become a common method for constructing compilers. Significant progress in tool improvement has been made, and is reported elsewhere[1,2,3,4,5]. Managing the complexity of the generation process is a configuration control problem, and this paper presents our efforts in that area.

We have built a compiler construction system called *Eli*†, which employs off-the-shelf tools that generate compiler components from formal specifications. Eli accepts a non-redundant specification of the desired compiler, derives appropriate tool inputs from that specification, applies the tools, and then combines the generated modules or fragments with code from a library to produce a complete compiler.

We are not concerned here with the tools employed by Eli, but rather with the management of the specifications, tools and partial products. Nevertheless, we begin by outlining the process of compiler construction, listing the specification mechanisms normally used for tool inputs and giving references to more detailed discussions of their implementation. Then we present Eli's configuration control problem, and indicate the facilities needed to solve it. Finally, we discuss the implementation of Eli using Odin[6] and RCS[7].

2. A Brief Outline of Compiler Construction

A rather stable gross design for a compiler has evolved from experience during the last twenty years. This means that compilers for a wide variety of source languages and target machines can be described by supplying "parameters" to a single design model. The parameters consist, for the most part, of non-procedural specifications that define the behavior of compilation subtasks. Many different descriptive techniques are used for these specifications, as illustrated by Table 1.

The relationships among the subtasks listed in Table 1 can be described non-procedurally by an attribute grammar[17,9]. Mechanisms for dealing with the individual subtasks are also well-known; references for the various descriptive techniques are given in the second column, and the necessary algorithms can be found in most texts on compiler construction[9,18]. The third column of Table 1 indicates how a compiler designer normally obtains a description of the corresponding subtask. Several of the subtasks, such as the mapping of identifiers to internal representation and the analysis of the source language's scope rules, are carried out by standard algorithms that can be used unchanged. These algorithms are embodied in abstract data types whose operations are invoked at appropriate points during the compilation by using them to compute attributes.

Table 1
Subtasks of a Typical Compiler

Subtask	Descriptive technique	User action
Scanning	Regular expression[8,3]	Adapt
Identifier table	Abstract data type[2]	—
Denotation conversion	Abstract data types[9]	Adapt
Parsing	Context-free grammar[10,11]	Create
Scope analysis	Abstract data type[12,13]	—
Type analysis	Patterns[14]	Create
Storage mapping	Abstract data type[9]	—
Operation mapping	Patterns[15]	Create
Control mapping	Schemata[14]	Create
Peephole optimization	Register transfers[16]	Adapt
Assembly	Formats[9]	Adapt

Since a wide variety of languages have similar lexical structures, descriptions of the scanner and the conversion of denotations (constants such as "1234") can usually be adapted from previously-existing descriptions. Register transfers and the formats that govern the treatment of the target machine can also be adapted in many cases. Descriptions of parsing and type analysis depend sensitively upon the source language, and the descriptions of operation and control mapping define the relationship between the source language and the target machine. These descriptions almost always must be created anew for each compiler, but there will be many similarities with existing descriptions.

Given a coherent model of a compiler and a set of techniques to describe the various components of that model, the obvious next step is to automate the construction of the compiler itself[19]. Tools have been developed for generating most of the components from formal specifications (see the references quoted in Table 1). Moreover, programs to control the components' interaction can be generated from attribute grammars[20,21]. A complete compiler can be created if the outputs of all of these tools can be combined into a smoothly-functioning product. A recent paper describing the ACORN system[14], developed at Brown University in the early 1980's, gives a good overview of the problem of creating a complete compiler.

2. The Configuration Control Problem

Eli's configuration control problem can be briefly characterized as follows:

- There is a fixed set of relationships among specifications, tools and partial products that define the way in which a compiler must be built. In the terminology of DSEE[22], these relationships constitute the *system model*.
- The system can manufacture a large number of products, many of which are are important but rarely-used diagnostic aids.
- Consistency can only be checked by testing the specifications. The consistency of the products must be guaranteed by guaranteeing a consistent derivation; no test for consistency can be applied to the product.

† After Eli Whitney, who was the first US manufacturer to make extensive use of interchangeable parts.

- A "version" is defined by a particular set of specifications in association with a particular set of parameters to the generation process. It is necessary to be able to re-generate a specific version.
- Several people may be working on a single compiler. In general they will be working on different parts of the specification, and each will want to have access to the latest versions of the others' work.

The system model for Eli can be described by a *derivation graph* — an acyclic, bipartite graph whose node classes are objects and processes respectively[23]. Arcs from object nodes to process nodes indicate that the object is accessed by the process, while arcs from process nodes to object nodes indicate that the object is created by the process. Every process node in a derivation graph must have both predecessors and successors; an object node must have either predecessors or successors or both. A variety of data formats is modeled in the derivation graph by associating a *type* with each object. Two objects have the same type if and only if they contain the same kind of information and have identical formats.

Figure 2 is a derivation graph that describes a simplified version of Eli's system model. Many products are omitted, standard compilers and the linker are not explicit, and internal derivations carried out by the tools have been suppressed. The omitted portions of the graph do not introduce any new configuration control problems. Rectangular boxes represent tools, oval boxes represent specifications, and unboxed text represents derived objects. Currently Eli does not include tools that process peephole optimization and assembly specifications, so these subtasks are implemented by abstract data types. Also, type analysis is currently handled within the attribute grammar.

The dotted area at the top of Figure 2 delimits the part of the derivation not under Eli's control. CAGT[24] is an interactive tool that establishes a relationship between the concrete syntax used to describe the input program as written and the abstract syntax used to describe its semantic structure. This relationship is captured by an additional object that Eli treats as an extra specification. During the derivation of the compiler, CAGT is used in a "reverse" batch mode to combine the concrete syntax with the connection points that describe the abstract tree structure. The result is a parsing grammar.

The content of a derived object in Figure 2 is completely determined by the contents of the specifications and the parameters supplied to the derivation. Thus it is theoretically possible to manufacture a product directly from the specifications whenever it is requested. In many cases, however, the manufacturing effort can be reduced by re-using derived objects. This is particularly important when several closely-related products are manufactured without changing the specifications or parameters.

Most configuration control systems maintain a cache of derived objects, and re-derive an object only when "necessary". The simplest rule for determining that re-derivation is necessary is that some predecessor of the object in the derivation graph has been changed since the object was last derived[25]. This rule sometimes leads to more re-derivations than necessary, and more complex rules have been proposed in special cases[26].

Cache placement and access strongly affect the properties of a configuration control system. The objects in the cache should be invisible to the user because derived objects that are not products are of no concern. This is a simple application of the principle of modularity. On the other hand, the cache itself should be a visible object associated with a particular project. People working on the project should be able to share a particular cache, thereby making the results of their work available to each other.

4. Implementation

Eli is implemented as a set of off-the-shelf compiler construction tools managed by Odin[6]. Odin accepts a request for a certain product, carries out the steps necessary to obtain that

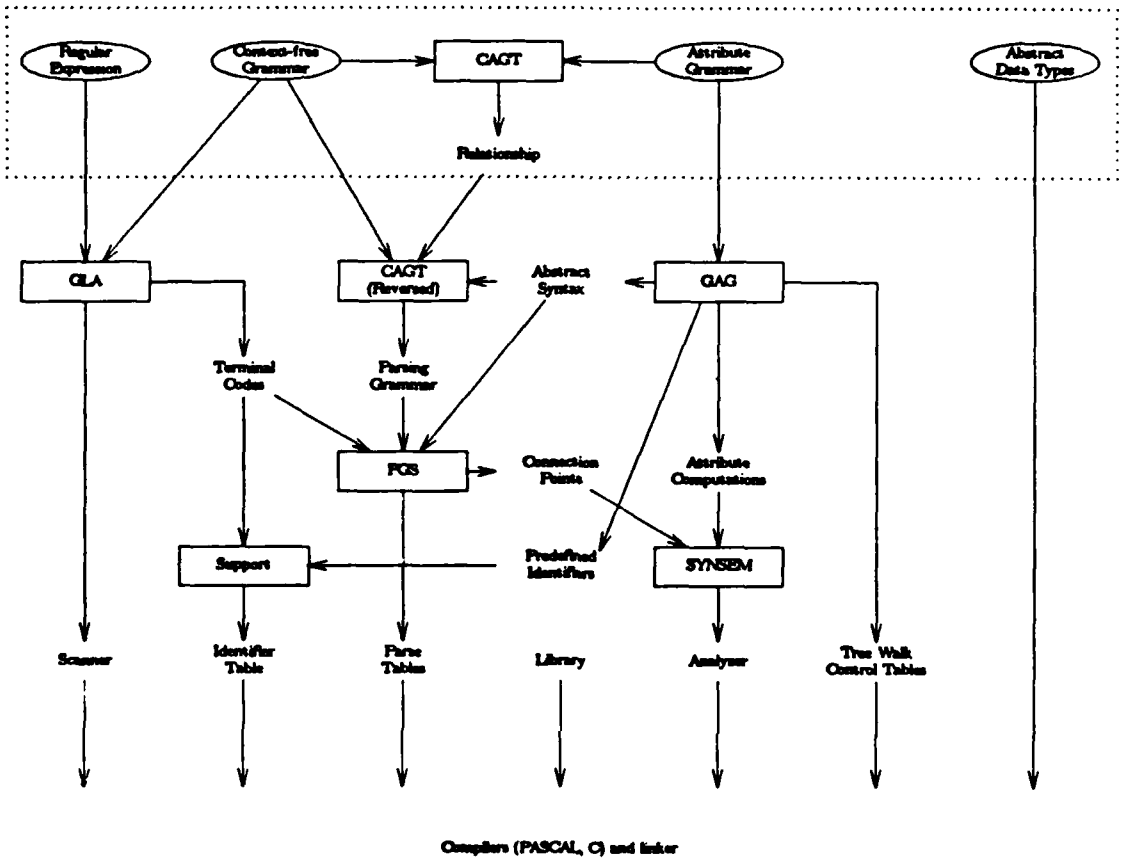


Figure 2
Simplified Derivation Graph for Eli

product, and delivers the result to the requestor. It maintains a cache into which it stores *every* derived object that it constructs. If a requested object is not in the cache, Odin uses the derivation graph to decide how to construct that object. It then constructs the object by following a path in the derivation graph, invoking the necessary processes and storing intermediate objects in the cache.

The objects manipulated by Odin are normal Unix files. Each process node in the derivation graph is associated with a Unix shell script. When a process is to be run, Odin modifies the shell script by filling in the names of the files that represent the associated objects, and then invokes the normal shell to execute the modified script.

A shell variable in the user's environment is assumed to specify the path name of a directory containing the derivation graph and subdirectories for the process nodes' shell scripts and the cache. There can, of course, be many such directories that link to the same derivation graph and shell scripts but have separate caches. Each member of the project group can therefore use a private cache or one shared among a set of colleagues, as appropriate. In any case, the

products requested are guaranteed to be consistent with the specifications used to derive them.

Because Odin objects are arbitrary Unix files, and because processes are defined by arbitrary Unix shell scripts, Odin provides the necessary flexibility to accommodate off-the-shelf tools. Odin is also in the public domain, and is easily transported to any Unix system. Unix itself simplifies the task of building processes that split, filter and merge data.

Users interact with Eli via the Odin query language. The derivation graph that defines Eli allows the user to derive any one of a number of objects from a specification, and these derivations may be parameterized. A "query" is a request for a particular derivation. Each query begins by stating the specification from which the derivation is to start, continues with a sequence (possibly empty) of keyword parameters and their values, and concludes with the object to be derived. Keyword parameters are introduced by the character "+", and an object to be derived is introduced by the character ":". For example, a Pascal compiler called "mypc" might be created from a specification called "pascal.g" by the following query:

```
pascal.g + name=mypc : compiler
```

Here the desired object is of type "compiler", and "mypc" is to be used as the value of the keyword parameter "name" in the derivation.

Queries can be obtained from files or presented interactively, and there is a history mechanism that allows one to modify and reuse interactive queries. Each query must specify a single object from which a product is to be derived. This object may be a list of file names, however. An object of type ".g" is a list of the file names that contain all of the specifications in the dotted part of Figure 2 except the abstract data types. Abstract data types are introduced into the derivation as values of keyword parameters, because the set of abstract data types varies from one compiler to the next.

We use RCS[7] to provide version control for the specifications. If the files mentioned by the ".g" object from which the derivation begins are not available in the working directory, they are sought in a subdirectory of the working directory named "RCS". When no parameters have been given, the latest revision is accessed. A version parameter can be specified in the query, and will be passed to RCS if it is present. Thus all of the version naming facilities of RCS are available.

An Odin query corresponds to DSEE's configuration thread[22]: Objects derived from queries with different parameter values are considered by Odin to be potentially distinct. (When Odin derives a new instance of an object it compares that instance with any existing instance. If they are identical, it marks the object as unchanged by the derivation.) Thus a query that contains a version parameter can be saved and used to regenerate a particular version of any product at any time.

5. Conclusions

Eli has been successful in improving the productivity of compiler constructors. One graduate student at the University of Colorado constructed a complete compiler for Whetstone ALGOL[27] in six weeks. He had previously taken the graduate compiler construction course (which was not based on tools) and had had industrial compiler experience. He had studied the ALGOL 60 Report in a graduate programming languages course, but had no other experience with ALGOL. His compiler did not follow the design given in Randell and Russell's book, since that design does not fit the Eli model, so the six weeks included a complete redesign. An experienced software engineer, who had worked with the Whetstone compiler on the ICL KDF9, estimated that he could carry out a new implementation of Randell and Russell's compiler in twelve weeks[28].

Odin was an appropriate mechanism to use in constructing Eli. The concept of a derivation graph captures the designer's intuition precisely, making it easy to understand and modify. Odin's implementation, although somewhat obscure in certain details, is flexible enough to

incorporate off-the-shelf components. This greatly simplifies the development of the system. It also increases the system's lifetime, because individual tools can be replaced by better technology without seriously disturbing the entire environment.

Acknowledgement

This work was partially supported by the Army Research Office under contract DAAL 03-86-K-0100 and by the Office of Naval Research under contract N00014-86-K-0204.

References

1. W. M. Waite and L. R. Carter, 'The Cost of a Generated Parser', *Software—Practice & Experience*, **15**, 221-239 (March 1985).
2. W. M. Waite, 'The Cost of Lexical Analysis', *Software—Practice & Experience*, **16**, 473-488 (May 1986).
3. V. P. Heuring, 'The Automatic Generation of Fast Lexical Analyzers', *Software—Practice & Experience*, **16**, 801-808 (September 1986).
4. R. W. Gray, 'Generating Fast, Error Recovering Parsers', M.S. Thesis, Department of Computer Science, University of Colorado, Boulder, CO, April 1987.
5. M. L. Hall, 'The Optimization of Automatically Generated Compilers', Ph.D. Thesis, Department of Computer Science, University of Colorado, Boulder, CO. 1987.
6. G. M. Clemm, 'The Odin System - An Object Manager for Software Environments', Ph.D. Thesis, Department of Computer Science, University of Colorado, Boulder, CO, 1986.
7. W. F. Tichy, 'RCS — A System for Version Control', *Software—Practice & Experience*, **15**, 637-654 (July 1985).
8. M. E. Lesk, 'LEX — A Lexical Analyzer Generator', Computing Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.
9. W. M. Waite and G. Goos, *Compiler Construction*, Springer Verlag, New York, NY, 1984.
10. S. C. Johnson, 'Yacc — Yet Another Compiler-Compiler', Computer Science Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, July 1975.
11. P. Dencker, K. Dürre and J. Heuft, 'Optimization of Parser Tables for Portable Compilers', *ACM Transactions on Programming Languages and Systems*, **6**, 546-572 (October 1984).
12. J. V. Guttag, 'Abstract Data Types and the Development of Data Structures', *Communications of the ACM*, **20**, 396-404 (June 1977).
13. S. P. Reiss, 'Generation of Compiler Symbol Processing Mechanisms from Specifications', *ACM Transactions on Programming Languages and Systems*, **5**, 127-163 (April 1983).
14. S. P. Reiss, 'Automatic Compiler Production: The Front End', *IEEE Transactions on Software Engineering*, **SE-13**, 609-627 (June 1987).
15. R. Landwehr, H. Jansohn and G. Goos, 'Experience With an Automatic Code Generator Generator', *SIGPLAN Notices*, **17**, (June 1982).
16. J. W. Davidson and C. W. Fraser, 'Automatic Generation of Peephole Optimizations', *SIGPLAN Notices*, **19**, 111-116 (June 1984).
17. K. Räihä, 'Bibliography on Attribute Grammars', *SIGPLAN Notices*, **15**, 35-44 (March 1980).
18. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers*, Addison Wesley, Reading, MA, 1986.

19. S. Rosen, 'A Compiler-Building System Developed by Brooker and Morris', *Communications of the ACM*, **7**, 403-414 (July 1964).

20. K. Räihä, M. Saarinen, E. Soisalon-Soininen and M. Tienari, 'The Compiler Writing System HLP (Helsinki Language Processor)', Report A-1978-2, Department of Computer Science, University of Helsinki, Helsinki, Finland, March 1978.

21. U. Kastens, B. Hutt and E. Zimmermann, *GAG: A Practical Compiler Generator*, Springer Verlag, Heidelberg, 1982.

22. D. B. Leblang and R. B. Chase, Jr., 'Computer-Aided Software Engineering in a Distributed Workstation Environment', *SIGPLAN Notices*, **19**, 104-112 (May 1984).

23. E. Borison, 'A Model of Software Manufacture', in *Advanced Programming Environments*, vol. 244, R. Conradi, T. M. Didriksen and D. H. Wanvik, (eds.), Springer Verlag, Heidelberg, 1986.

24. A. Bahrami, 'CAGT — An Automated Approach to Abstract and Parsing Grammars', MS Thesis, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO, 1986.

25. S. I. Feldman, 'Make — A Program for Maintaining Computer Programs', *Software—Practice & Experience*, **9**, (April 1979).

26. W. F. Tichy, 'Smart Recompilation', *ACM Transactions on Programming Languages and Systems*, **8**, 273-291 (July 1986).

27. B. Randell and L. J. Russell, *ALGOL 60 Implementation*, Academic Press, London, 1964.

28. B. K. Haddon, '', Personal Communication, December 1986.

END

DATE

FILMED

8-88

DTIC