**A Uniform Approach for Compiler.**

**1Sandeep Giri ,2Sagar Choudhary ,3Chunchun Singh and 4Rajat Verma**

1Student(B.tech 5th sem ) Department of Computer Science Engineering

Dronacharya College of Engineering, Gurgaon,India.

2Student(B.tech 5th sem ) Department of Computer Science Engineering

Dronacharya College of Engineering, Gurgaon,India.

3Student(B.tech 5th sem ) Department of Computer Science Engineering

Dronacharya College of Engineering, Gurgaon,India.

4 Student(B.tech 5th sem ) Department of Computer Science Engineering

Dronacharya College of Engineering, Gurgaon,India.

**1.Abstract:** - Designed to provide good system programming in semantic and syntactic is the process of compiler design. Any program written in a high level programming language must be translated to object code before going to be executed. Compiler is the need to design and connectivity between the hardware and software process, learning English grammar provides a precise way to specify the syntax and meaning of a language to speak and write proper English. A grammar in compiler is a set rule that specify how sentences can be structured with the terminals, non-terminals and the set of productions. Code generation for embedded processors is the design of efficient compilers for target machines, we describes the application specific features in a compiler and backend design that accommodates these features by means of compiler register allocation and supports the embedded

systems and media applications. This analysis presents the techniques of compiler design and also design of network processor and embeds system, compiler not only translates the information can be used for the processor design.

**2.Keywords**:- Specialization- pursuing a particular line of work, Fidelity- strict observance of promises, Immutable- unchangeable, Trivia- inconsequential, Computation- an act or process.

**3.INTRODUCTION:-**

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly).A program for a computer must be built by combining some very simple commands into a program in what is called machine language. Since this is a tedious and error prone process most programming is, instead, done using a high-level programming language. Programs are usually written in high level code, which has to be converted into machine code for the CPU to execute it. This conversion is done by either a compiler (or a linker) or an

interpreter, the latter generally producing binary code, machine code, that can be processed to be directly executable by computer hardware but compilers will proceed usually by first producing an intermediate binary form called object code. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler and interpreter come in.

**4. COMPILER:-**

A compiler [1] translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes. Using a high level language for programming has a large how fast programs can be developed. The main reasons for this are:

• Compared to machine language, the notation used by programming languages closer to the way humans think about problems.

• The compiler can spot some obvious programming mistakes.

• Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.

• Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.

On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler will,

How ever, be able to get very close to the speed of hand-written machine code when translating well-structured programs.

**5. THE PHASES OF COMPILER**

Since writing a compiler is a nontrivial task, it is a good idea to structure the work. A typical way of doing this is to split the compilation into several phases with well-defined interfaces. Conceptually, these phases operate in sequence (though in practice, they are often interleaved), each phase (except the first) taking the output from the previous phase as its input. It is common to let each phase be handled by a separate module. Some of these modules are written by hand, while others maybe generated from specifications. Often, some of the module scan be shared between several compilers. A common division into phases is described below. In some compilers, the ordering of phases may differ slightly, some phases may be combined or split into several phases or some extra phases may be inserted between those mentioned below.

**I .Lexical analysis:** This is the initial part of reading and analyzing the program text. The text is read and divided into tokens ,each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.

**II. Syntax analysis:** This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure(called the syntax tree) that reflects the structure of the program. This phase is often called parsing.

**III. Type checking :** This phase analyses the syntax tree to determine the program violates certain consistency requirements ,e.g., if a variable is used but not declared or if it is used in acon text that does not make sense given the type of the variable ,such as trying to use a boolean value as a function pointer.

**IV. Intermediate code generation:** The program is translated to a simple machine independent intermediate language. The symbolic variable names used in the intermediate code are translated to numbers, each of which corresponds to a register in the target machine code.

**V. Machine code generation:** The intermediate language is translated to assembly language (a textual representation of machine code) for specific machine architecture.

**VI. Assembly and linking:** The assembly-language code is translated into binary representation and addresses of variables, functions, etc., are determined.

The first three phases are collectively called the frontend of the compiler and the last three phases are collectively called the back end. The middle part of the compiler is in this context only the intermediate code generation, but this often includes various optimizations and transformations on the intermediate

code. Each phase, through checking and transformation, establishes stronger invariants on the things it passes on to the next, so that writing each subsequent phase is easier than if these have to take all the preceding into account. For example, the type checker can assume absence of syntax errors and the code

generation can assume absence of type errors. Assembly and linking are typically done by programs supplied

by the machine or operating system vendor, and are hence not part of the compiler itself, so we will not further discuss these phases in this book.



Figure 1:Steps involve in code Execution

**6. Structure of a compiler:-**

**Compilers** bridge source programs in high level languages with the underlying hardware.

A compiler requires :-

1) determining the correctness of the syntax of programs

2) generating correct and efficient object code

3) run-time organization

4) formatting output according to assembler and/or linker conventions.

A compiler consists of three main parts:

The frontend,

The middle-end

The backend.

I.The **front end :**checks whether the program is correctly written in terms of the programming language syntax and semantics.

In this:-

1) legal and illegal programs are recognized.

2) Errors are reported.

3) Type checking is also performed by collecting type information.

The frontend then generates an *intermediate representation* or *IR* of the source code forprocessing by the middle-end.

II. The **middle end:** is where optimization takes place. Typical transformations for optimization are removal of useless or

unreachable code, discovery and propagation of constant values, relocation of computation to a less frequently executed place (e.g., out of

a loop), or specialization of computation based on the context. The middle-end generates another IR for the following backend. Most

optimization efforts are focused on this part.

III. The **back end:** is responsible for translating the IR from the middle-end into assembly code .The target instruction(s) are chosen for each

IR instruction. Register allocation assigns processor registers for the program variables where possible. The backend utilizes the hardware by figuring out

how to keep parallel execution units busy ,filling delay slots, and so on.

**7.Need for Design Principles:** Furthermore as partial evaluators treat richer and richer languages their size and complexity increase drastically. Indeed programs written in realistic languages like C expose a very wide variety of situation where partial evaluation can be applied\_ As a result there is now a clear need to propose design principles to structure the added complexity of the resulting

partial evaluators Compile Time Specialization is Limiting . When studying components from real software systems it becomes apparent that exclusively specializing programs at compile time is limiting In fact there exist numerous invariants that are not known until run time and can yet be used for extensive specialization\_ This situation occurs for example when a set of procedures implements session\_

oriented transactions\_ When a session is opened many pieces of information areknown but only at run time\_ They could be used to specialize the procedures which perform the actual transactions\_ Then when the session is closed theinvariants would become invalid􀀀 therefore the specialized procedures can be eliminated.\_

**8. Design Methodology:** Microprocessor design uses CAD tools provide a starting point in the design process. Existing processor designs provide an architectural reference point from which design modification can be made desirable architectural features. The most accurate method of architectural assessment involves circuit level timing simulation of full processor layout and cycle level simulation of full applications based on optimized compiled code.

This architecture referred to as an architecture instance by evaluating its performance on a suite of applications using a mapping by a compiler to generate the assembly code performing analysis evaluate the resulting code. The implications of the results can be used for iterative improvements to the architecture instance mapping or applications. For our purpose we desire a media processor that I high-level language programmable without requiring special libraries or iterative improvements by the programmer for performance. The figure describes the design process at each level of the design space exploration, abstraction pyramid represent the early stages of the design space exploration .Evaluation environment for this processor was provided by the IMPACT compiler is ideal because it supplies not only an aggressive ILP compiler but also the simulation environment and performance analysis tools necessary for a design space exploration using the design methodology. The basic compiler method enables the primary optimization of three paths

1. Classical optimizing and procedure in lining

2. Superblock includes all optimizations in classical and adds the superblock

optimization and loop unrolling

3. Hyper block includes all optimizations in superscalar and adds the hyper block

optimization.

**9.Parsing Application**

**I. Parsing Application:**

Parsing can be defined as a process of analysing a text which contains a sequence of tokens to determine its grammatical structure in given grammar. One the source code is syntactically valid the compiler has generated into abstract syntax tree or syntax directed translation of the

source code.

Classical parsers conventionally accept a context free language defined by a context free, for each program parser does produce a phrase structure referred to as an abstract syntax tree. Parse including error stabilization and AST constructors can be generated from context free grammars for parsers. Classical parsing techniques may be applied as long as program conforms to the syntax of a programming language however can be assumed in general as programs to analyze can be incomplete erroneous or conform to a dialect of the language. Fuzzy parsing was designed to efficiently develop parsers by performing the analysis on selected parts of the source instead of the whole input. It is specified by a set of fuzzy context free sub grammars each with their own axioms does not require strict adherence to a language grammar. It scans for instances of the axioms and then parses according to the grammar makes parsing more robust since it ignores source fragments including missing parts errors and deviations that subsequent analyses abstract from anyway. Fuzzy parsers Delphi XPG utilise a context fuzzy parsing technology for building across identifiers, OPARIis source to source translation tool which automatically adds all necessary calls to the time measurement which allows to collect runtime performance data of fortran, C, C++applications.

Using Natural Language processing technique can be applied to formal language dependency structure is one way of representing the syntax of natural language. This technique automatically generates the language specific information extractor using machine learning and training of a generic parsing instead of explicitly specifying the information extractor using grammar and transformation rules. Android Application Development web service request and response uses three types

of parsing xml DOM Parser PULL Parser SAX Parser. Android provides a library that contains classes used to parse xml by constructing a document and matching each node to parse the info to parse with DOM parser as shown below. Void parse By DOM(String response) throws Parser Configuration Exception, SAX Exception,

IO Exception{Person person=new Person();

Document Builder Factory dbf = Document Builder Factory. New Instance ();

Document Builder db = dbf. New Document Builder ();

Document doc = db. parse(new Input Source (new String Reader (response)));

// normalize the document

doc. get Document Element ().normalize();

// get the root node

Node List node List = doc. Get Elements By Tag Name ("person");

Node node=nodeList. Item(0);

// the node has three child nodes

for (int i = 0; i < node.getChildNodes()

getLength();

i++

{

Node temp=node.getChildNodes().item (i);

if(temp.getNodeName().equalsIgnoreCase("firstname")){

person.firstName=temp.getTextContent ();

}

else if(temp.getNodeName().equalsIgnoreCase("lastname")){

person.lastName=temp.getTextContent ();

}

else if(temp.getNodeName().equalsIgnoreCase("age")){

person.age=Integer.parseInt (temp.getTextContent ());

}

}

Log.e ("person", person.firstName+ " "+person.lastName+"

"+String.valueOf(person.age));

}

It retrieves the correct information but the user needs to familiar with the xml structure so that we know the order of each xml. Android provides org.xml.sax package that has provide the event driven SAX parser to parse the previous response with SAX parser have to create a

class extending Default Handler. Start Document() invoked when the xml document is open there we can initialize any

member variables. Start Element () invokes when the parser encounters a xml node here we can initialize specific instances of our object. end Element () invoked when the parser reaches the closing of a xml tag here the element value would have been completely. Characters () this method is called when the parser reads characters of a node value. The method optJSONArray, optString, optInt instead of using getString, getInt because theopt methods return empty strings or zero integers if no elements are found. Parse platform provides a complete background solution for mobile application, storing data on parse is built around the Parse Object contains key value pairs of JSON compatible data.

**7.Applications:-**

Finding code fragments that look like assertions typifies the use of lcsc to find patterns in C#. Elaborations of this usage are used for code-auditing tools, for example. Pattern matching on an AST instead of text makes it easy to consider context and to fine tune matches to avoid voluminous output. It’s tempting to design a little language for specifying tree patterns, perhaps similar to AWK or to the languages used in tree-based code-generator generators [5]. But for many pattern-based applications, the C# code is so short

that more concise specifications are not needed.XML is an increasing popular way to exchange data, and there are numerous tools available for editing and viewing XML. The XML visitor emits an AST as XML for consumption by XML-based tools or external compilation tools that accept XML. C# includes extensive support for reflection, which makes it possible to discover class information during execution. The 70-line XML visitor uses reflection to discover the details of the AST classes necessary to emit XML and is thus automatically upward compatible with future additions to the AST vocabulary. The visitor architecture helps write meta programming tools, e.g., tools that write programs. The .NET platform includes a tree-based API for creating programs, typically those used in web services applications .This API defines a language-independent code document object model, also known as Code DOM. It’s possible, for example, to build a Code DOM tree and pass it to C#, Visual Basic, or to any language that offers a `code provider’ interface. A common approach to writing Code DOM applications is to write, say, C# source code and translate it by hand into the API calls that build the Code DOM tree for that C# code. The lcsc code dom visitor automates this process: Given a C# program *P*, it emits a C# program that ,when executed, builds the Code DOM tree for *P*.

The source visitor is similarly useful: It emits C# source code from an AST. When coupled with visitors that modify the AST, source provides a source-to-source transformation facility. As detailed in the next section, source is useful for C# language extensions that can be modeled in itself. It’s also useful for writing code reorganization tools. A simple example is sort members, which alphabetizes the fields and methods in all of the classes in its input program.

**8.The Tasks of a Compiler:-**

A compilation is usually implemented as a sequence of transformations (Lk; TL), where SL is the source language and TL is the target language. Each language Li is called an intermediate language. Intermediate languages are conceptual tools used in decomposing the task of compiling from the source language to the target language. The design of a particular compiler determines which (if any) intermediate language programs

actually appear as concrete text or data structures during compilation Any compilation can be broken down into two major tasks:

\_ Analysis: Discover the structure and primitives of the source program, determining its meaning.\_ Synthesis: Create a target program equivalent to the source program. This breakdown is useful because it separates our concerns about the source and target

languages. The analysis concerns itself solely with the properties of the source language. It converts

the program text submitted by the programmer into an abstract representation embodying the essential properties of the algorithm. This abstract representation may be implement in many ways, but it is usually conceptualized as a tree. The structure of the tree represents

the control and data aspects of the program, and additional information is attached to the nodes to describe other aspects vital to the compilation. the general characteristics of source languages, pointing out the properties relevant for the compiler writer. illustrates the general idea with an abstraction of the algorithm describes the control and data of the algorithm by means of the 'kth

descendant of' relation. For example, to carry out the algorithm described by a subtree4 Introduction and Overview

Node Additional corresponding declaration name type of the variable exp type of the expression value) Additional information about the source program Node Additional Information name corresponding data location if address of code to carry out the else part

while address of the expression evaluation codec) Additional information about the target program Figure 1.3: An Abstract Program Fragment rooted in a while node we evaluate the expression described by the sub tree that is the descendant of the while node. If this expression yields true then we carry out the algorithm described by the sub tree that is the second descendant. Similarly, to evaluate the

expression described by an expression sub tree, we evaluate the \_rst and third descendant sand then apply the operator described by the second descendant to the results. The algorithm of Figure 1.1a is not completely characterized by Figure 1.3a. Information

must be added (Figure 1.3b) to complete the description. Note that some of this information(the actual for each idn) is taken directly form the source text. The remainder is obtained by processing the tree. For example, the type of the expression value depends upon

the operator and the types of the operands. Synthesis proceeds from the abstraction developed during analysis. It augments the tree

by attaching additional information (Figure 1.3c) that the source-to-target mapping discussed in the previous section. For example, the access function for the variable i in Figure 1.1a would become the address of data location I according to the mapping M assumed

by Figure 1.2. Similarly, the address of the else part of the conditional was represented by the label SUBI. discusses the general characteristics of machines, high lighting properties that are important in the development of source-to-target mappings.

Formal of the source language and the source-to-target mapping determine the structure of the tree and the computation of the additional information. The compiler simply implements the indicated transformations, and hence the abstraction illustrated in Figure 1.3

forms the basis for the entire compiler design. we discuss this abstraction in detail, considering possible intermediate languages and the auxiliary data structures used in transforming between them. Analysis is the more formalized of the two major compiler tasks. It is generally broken1.3 Data Management in a Compiler 5down into two parts, the structural analysis to determine the static structure of the source

program, and the semantic analysis to \_x the additional information and check its consistency. Chapter 5 summarizes some results from the theory of formal languages and shows how they are used in the structural analysis of a program. Two subtasks of the structural analysis are

on the basis of the particular formalisms employed: Lexical analysis (Chapter 6)deals with the basic symbols of the source program, and is described in terms of syntactic analysis, or parsing deals with the static structure of the

program, and is described in terms of pushdown automata. extends the theoretical treatment of Chapter 5 to cover the additional information attached to the components of the structure, and applies the resulting formalism (attribute grammars) to semantic analysis.

there is little in the way of formal models for the entire synthesis process, although algorithms for various subtasks are known. We view synthesis as consisting of two distinct subtasks, code generation and assembly. Code generation (Chapter 10) transforms the abstract source program appearing at the analysis/synthesis interface into an equivalent target machine program. This transformation is carried out in two steps: First we map the algorithm from source concepts to target concepts, and then we select a sequence of target

machine instructions to implement that algorithm. Assembly resolves all target addressing and converts the target machine

instructions into an appropriate output format. We should stress that by using the term 'assembly' we do not imply that the code generator will produce symbolic assembly code for input to the assembly task. Instead, it delivers an internal representation of target instructions

in which most addresses remain unresolved. This representation is similar to that resulting from analysis of symbolic instructions during the \_rst pass of a normal symbolic assembler. The output of the assembly task should be in the format accepted by the standard link editor

or loader on the target machine .Errors may appear at any time during the compilation process. In order to detect as

many errors as possible in a single run, repairs must be made such that the program is consistent, even though it may not reect the programmer's intent. Violations of the rules source language should be detected and reported during analysis. If the source algorithm

uses concepts of the source language for which no target equivalent has been implementation, or if the target algorithm exceeds limitations of a interpreter (e.g. requires more memory than a computer provides), this

should be reported during synthesis. Finally, errors must be reported if any storage limits ofthe compiler itself are violated.

In addition to the actual error handling, it is useful for the compiler to provide extra information for run-time error detection and debugging. This task is closely related to error handling, and both are discussed in.A number of strategies may be followed in an attempt to improve the target program relative to some measure of cost. (Code size and execution speed are typical cost

measures.) These strategies may involve deeper analysis of the source program, more complex mapping functions, and transformations of the target program. We shall treat the our discussions of analysis and code generation respectively; the third is the subject of

1.3 Data Management in a Compiler As with other large programs, data management and access account for many of the problems

to be solved by the design of a compiler. In order to control complexity, we separate the6 Introduction and Overview

functional aspects of a data object from the implementation aspects by regarding it as an instance of an abstract data type. (An abstract data type is by a set of creation ,assignment and access operators and their interaction; no mention is made of the concrete

implementation technique.) This enables us to concentrate upon the relationships between tasks and data objects without becoming enmeshed in details of resource allocation the machine upon which the compiler is running (the compiler host) rather than the

problem of compilation .A particular implementation is chosen for a data object on the basis of the relationship

between its pattern of usage and the resources provided by the compiler host. Most of the basic issues involved become apparent if we distinguish three classes of data:

\_ Local data of compiler tasks

\_ Program text in various intermediate representations

\_ Tables containing information that represents context-dependence in the program text

Storage for local data can be allocated statically or managed via the normal stacking mechanisms of a block-structured language. Such strategies are not useful for the program text, however, or for the tables containing contextual information. Because of memory limitations,

we can often hold only a small segment of the program text in directly-accessible storage. This constrains us to process the program sequentially, and prevents us from representing it directly as a linked data structure. Instead, a linear notation that represents a

traversal of the data structure is often employed. Information to be used beyond the immediate vicinity of the place where it was obtained is stored in tables .Conceptually, this information is a component of the program text; in practice it often

occupies data structures because it access patterns. For example, tables must often be accessed randomly. In some cases it is necessary to search them, a process that may require a considerable fraction of the total compilation time. For this reason we do not

usually consider the possibility of spilling tables to. The size of the program text and that of most tables grows linearly with the length of

the original source program. Some data structures (e.g. the parse stack) only grow with the complexity of the source program. (Complexity is generally related to nesting of constructs such as procedures and loops. Thus long, straight-line programs are not particularly complex.)

of bounds on the size of any of these data structures leads automatically to restrictions on the class of translatable programs. These restrictions may not be human programmer but may seriously limit programs generated by pre-processors.

6 Summary

We were motivated to build an object oriented compiler to allow for user control of the compilation process to customize both implementation and semantics. The natural granularity of customizability for compilation is not a unit of program text, but rather a group of data values and the program locations that will interact with them. We create a new kind of compile-time meta object, the compilation contract, that corresponds to

such units, and which allows user adjustments to be expressed locally yet have their consequences takeect consistently. the usage, the requirements etc. Also it can be said that some phases of compiler like optimization [7] which the interpret terlacks can be worked upon and can be included in the

interpreter to get the optimized results with low space us ageand greater efficiency. The approach is based on the follow in research i.e. trying to make a compiler for interpreting languages like java script, python, perl etc. The project aims atoptimizing the interpretation process. This includes significant

reduction in both size and time complexity .Most of the interpreted languages are in demand due to their simplicity but due to lack of optimization, they require comparatively large amount of time and space for execution. Also there is no method for code minimization. The code size is

larger than what actually is needed due to redundancy in the not encountered while compiling the code, so we aspire to design a Computer which adds optimization phases of a compiler in production pipeline of interpreted code and then produces the optimized source code for the interpreted language which will be optimized code in terms of running time

**FUTURE WORKS**

As the comparative analysis shows the proposed heuristic solution significantly enhances the efficiency of the code

optimization hence compiler optimization. This algorithm has a significant reduction in space complexity as well, as the

over head of coloring the graphs has been reduced. This enhances the performance of code and utilization memory of the system.

Future works include the application of parallelism fo compiler optimization along with graph coloring which will provide an effective technique of peephole optimization for compilers. Various parallelization techniques like task decomposition and domain decomposition models are suitable

for the evolution of compiler design [3].

**8. ACKNOWLEDGMENTS**

Our sincere thanks to all the faculty members, acquaintances and friends who helped us bring this work across.

**REFERENCES**

[1] Dragon Book-Code Optimizations

http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/20-Optimization.pdf. 2014.

[2] Compiler construction lecture notes

http://www.personal.kent.edu/~rmuhamma/Compilers/compnotes.html

[3] PyPyTechnology:http://doc.pypy.org/en/latest/interpreteroptimizations.

Html #introduction

[4] Self Optimizing AST:

http://www.christianwimmer.at/Publications/Wuerthinger12a/

[5] Optimizing Indirect Branch Prediction Accuracy in Virtual Machine

Interpreter https://www.scss.tcd.ie/David.Gregg/papers/toplas05.pdf

[6] Google Closure Compiler Introduction

https://developers.google.com/closure/?csw=1.

[7] Original slides from Computer Systems: A Programmer's Perspective

by Randal E. Bryant and David R. Randal E. Bryant and David R. O 'Hallaron O 'Hallaron “Code Optimization: Machine

Independent Optimizations”

[8] How JavaScript compilers work

http://creativejs.com/2013/06/the-race-for-speed-part-2-howjavascript-compilers-work/

[9] Sunil Kr. Singh, R. K. Singh, M.P.S. Bhatia “Performance Evaluation

of Hybrid Reconfigurable Computing Architecture over Symmetrical FPGA” IJESA, doi:10.5121/ijesa.2012.2312, Vol.2,No.3, page 107-116, September 2012.

[10] R. Agrawal. Alpha: An extension of relational algebrato express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, July1988.

[11] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[12] A.V. Aho and J.D. Ullman. *The theory of parsing ,translation, and compiling*. Prentice-Hall, Englewood Cliffs (NJ), 1972–73. Vol. I. Parsing. Vol II. Compiling.

[13] M. Alt, F. Martin, and R. Wilhelm. Generating Dataflow Analyzers with PAG. Technical Report A10-95, Universiti ¨at des Saarlandes, 1995.

[14] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with AddisonWesley,1989.

[15] I. Bethke, J.W. Klop, and R.C. de Vrijer. Descendantsand origins in term rewriting. *Information and Computation*,159(1-2):59–124, 2000.

[16] M. G. J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moon en ,P.A. Olivier, J. Scheerder, J.J. Vinju, E

[17] M. G. J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat ,editors, *Algebraic Methodology and Software Technology*

[18] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice& Experience*, 30:259–291, 2000.

[19] M. G. J. van den Brand, P. Klint, and C. Verhoef .Core technologies for system renovation. In K.G. Jeffery ,J. Kr´al, and M. Barto˘sek, editors, *SOFSEM’96:Theory and Practice of Informatics*, volume 1175 of *Lecture Notes in Computer Science*, pages 235–255.Springer-Verlag, 1996.

[20] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scanner less generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC’02)*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[21] G. Canfora, A. de Lucia, and G.A. Lucca. A system for generating reverse engineering tools: a case study of software modularisation. *Automated Software Engineering*,6:233–263, 1999.

[22] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow.TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[23] T.R. Dean, J.R. Cordy, A.J. Malton, and K.A. Schneider .Grammar programming in TXL. In *2nd International Workshop on Source Code Analysis and Manipulation(SCAM02)*. IEEE, October 2002.

[24] A. van Deursen. *Executable Language Definitions -Case Studies and Origin Tracking Techniques*. Ph Dthesis, University of Amsterdam, 1994.