An Efficient Approach toward a Flexible and General Knowledge Definition and Program Control Language System for a Synthesis-Planning Program

Wolf-Dietrich Ihlenfeldt

Institute of Organic Chemistry I, Technical University Munich, 85748 Garching, Germany[1]

Received November 24, 1993*

The concept and implementation of an embedded program control and knowledge encapsulation language pair is presented. It is efficient both in terms of execution speed and implementation effort. The WODCA synthesis-planning system uses this language set for a wide variety of tasks including complex rule-based operations such as strategic bond detection and merit evaluation of possible starting materials. Important features of the language pair include interactive run-time function development and modification, interactive debugging, and cross-compilation techniques for a maximum of code maintainability and execution efficiency.

1. CONTROLLING COMPLEX PROGRAMS

Programs which deal with complex chemical information generally require flexible control mechanisms to direct the flow of information within the program and to select the invoked computational modules. Simple techniques such as command line arguments, traditional FORTRAN style "control cards", and menu systems are not powerful enough to guide complex computations. This certainly does not imply that menu systems or program parameters are generally useless, quite the contrary. The optimal solution might well include mechanisms to map a complex custom command sequence to a menu button for easy invocation.

A variety of techniques have been employed to provide computational chemistry programs with some internal programmability. The most refined approaches in this domain probably have been implemented in synthesis-planning, reaction prediction, and structure elucidation programs. This is not surprising because these programs typically have a complex internal structure and complicated paths of information processing. In contrast quantum-mechanical programs are often comparatively simple and well structured in their information flow aspect and have few decision points. Synthesis design, reaction prediction, and structure elucidation programs on the other hand, may not only need controls to select the path of processing but complex rule systems which comprise the overall knowledge of the system.

Synthesis-planning, reaction prediction, and structure elucidation systems incorporate a large amount of more or less empirical knowledge which is conveniently encoded in rule sets. The external storage of these rule sets coded in a rule specification language adds flexibility to the system. Rule sets can then easily be adapted to specific problems. Maintenance becomes much easier. While in the programs described in the literature and described later in this paper there is often a clear distinction between rule languages and control languages (most implement only one of those), this distinction is artificial.

The rule system in the aforementioned class of programs is often not only concerned with the comparison of elementary chemical properties and structure data against rule set conditions but may itself perform some nontrivial computation of high-level structure data from low-level primitives. In this case the efficiency of implementation is a major contributor to the overall performance of the program system. Implementation issues are certainly no concern if every 30 min of execution time a module is selected on the basis of user-specified instructions, but rule-based systems often spend a majority of the total run time in the environment of the control/rule language.

An overview of the features of a sample of well-known programs from the literature gives an idea about the various popular approaches toward control and rule languages in chemistry-oriented programs. While there is an obvious trend toward continuously more modular and external information coding, this is not enforced, and therefore the list is not in historical order:

1. Systems such as the reaction prediction programs EROS5.0, EROS 5.2,[2] and CAMEO[3] and the synthesis planning program CHIRON[4,6] do not have any internal programmability to speak of. Some of these programs contain hundreds of rules and algorithmic tidbits to treat chemical structures which are all hard-coded. A recompilation is needed for every change in the rule base. Limited control may be possible via command line switches and keyword configuration files.

2. The PAIRS IR interpretation program[5] uses an extremely simple rule-encoding scheme which allows basic arithmetic operations, branches, and blocks. The rules are preparsed into binary tables.

3. The EROS6.0 reaction prediction program[7,8] uses external rules which are precompiled into a binary table file. This table is loaded by the main program before processing begins. The criteria used to control the applicability of a transformation or reaction are switch-type statements without high-level controls but extensive mathematical function support. In the synthesis planning sector, the recently added starting-material-directed retrosynthetic analysis of the LHASA program[9] uses the same switch-type approach to enable or disable criteria for the synthetic proximity merit function. This information is read from an ASCII file at run time.

4. The EROS6.1 system uses a rule language called VERGIL,[10] which supports branches, blocks, subroutines, and sophisticated mathematical functions for the reactivity evaluation. High-level constructs such as loops, local variables, and recursion are still missing. The rule files are parsed at run time and translated into a parse tree, which is executed when a rule is applied. The retrosynthetic rule languages of the early LHASA versions[11-13] and SECS[14,15] (*ALCHEM*) have a similarly rather limited scope, although the set of

* Abstract published in *Advance ACS Abstracts*, April 1, 1994.

0095-2338/94/1634-0872$04.50/0

features is obviously different (set expressions, but no floating point arithmetic, etc.). The rules are either read directly into internal tables or translated to FORTRAN source code BLOCK DATA statements, which are compiled into the program.

5. The most highly developed language for reaction prediction and retrosynthetic planning[16] is probably found in the later LHASA versions.[17] *CHMTRN* supports important features found in general-purpose languages such as loops, subroutines, and jumps in addition to the standard set of branches, blocks, and arithmetics.

The architectures of the programs mentioned so far still consider the rule base as an item controlled by the core program. Rules do not have dynamics of their own.

6. Programs such as MYCIN[18] for medical diagnosis or DENDRAL[19] for structure elucidation use more exotic rule and control language philosophies. Programs written in LISP and especially in AI languages such as PROLOG do not have a clear distinction between framework code and rule definitions. Rule execution is not necessarily initiated explicitly by some core routines but might be implicit, as in the case of the PROLOG inference engine.

However, all these control language implementations have a very limited scope. Either they encode rules or they control the program flow, but not both. The external functions and rules need to be coded completely before the program runs. As far as it can be concluded from the published material, none of the implementations support interactive development, i.e. writing, testing, and modifying functions at run time in an interactive environment. Debugging is also a major problem. It seems interactive debuggers for these languages are not available; even I/O functionality for paper traces is often lacking.

2. LANGUAGE REQUIREMENTS: THE WODCA SYSTEM

WODCA is an interactive synthesis-planning program which does not follow the traditional path of systematic exhaustive reaction or retro-reaction rule application.[20-22] Rather, WODCA tries to locate, in close interaction with an experienced chemist, suitable starting materials from large catalogs by the detection of *synthetically exploitable* similarities between starting materials and synthesis targets.[23] Novel similarity search algorithms were developed for this purpose. In case no direct relationship can be established, WODCA detects automatically or under cooperation with the user strategic bonds, which are cut separately or concurrently using a variety of mechanisms. This fragmentation and similarity detection procedures are repeated until a set of acceptable starting materials have been found. The result is a log file, which displays a tree structure of possible pathways. In contrast to classical reaction trees, this tree contains high-level information such as complete match lists and the generating search instructions, generalized fragments, similarity ratings, and the methods applied to generate them, and similar items.

WODCA needs to have an extendable, external rule and command language for the following tasks:

Command Scripts. These react to events and structural features of molecules encountered during the execution of batch files or command blocks.

Filters. These accept or reject molecules or ensembles of molecules for I/O (to/from files or data bases) and specific processing or editing tasks.

Strategic Bond Detection. This detects automatically strategic bonds in molecules and suggests methods to make or break them. This means that a bond is not simply marked as *strategic* but includes also the generation of pseudoappendix atoms on the fragments or the tagging of remote atoms as atoms to be eliminated when the bond is cut. Strategic bonds may be grouped into sets for synchronous reactions and have a number of auxiliary attached properties.

Merit Evaluation of Strategic Bond Break Operations. After breaking a single strategic bond or a set of them and adding pseudoatoms or eliminating $\alpha/\beta/\delta$ substituents, the fragments are subjected to another merit evaluation function, which checks for example if a decrease in molecular complexity was achieved.

Merit Evaluation of Precursor Mappings. This function type yields information about whether a starting material from a library maps reasonably well onto a target and whether the bonds which need to be manipulated both in the target and the library molecule are reactive and can be selectively attacked. The stereochemical fit is also examined here.

Detection of Strategic Bonds in Catalog Molecules. In cases where a large-scale degradation of starting materials is expected to be useful in a synthesis, bonds not only in the target but also in the entries of the starting material catalogs (or, for efficiency reasons, a prefiltered subset of the molecules in a catalog) can be marked as strategic, so a two-way approximation between target and precursor is achieved, with the ultimate goal of finding similarity that is clearly defined and can be exploited in a synthesis.

Due to the complexity of these functions, a number of features of the control/command language are indispensable:

External Storage. Function text must be separate from the main program and stored in simple ASCII text files which are readable to humans.

Compilation. A pure interpreter is too slow. On the other hand, a true compiler which produces machine code is too complex and not portable. So an intermediate-level representation of the functions is desirable.

Debugging. Interactive debugging which includes stepping, breakpoints, variable value examination and manipulation, and expression evaluation is a major requirement.

Interactive Development. It should be possible to edit and change the functions at run time. Multiple sets of functions must be available and should be chosen at run time, automatically or by user directives.

Extensive Set of Features. Full support for subroutines, functions, branches, various types of loops, local and global variables of different types, arrays, I/O, and recursion is mandatory. The possibility to create user-defined data types and modern computer language design concepts such as operator overloading are probably useful.

Small Core. While a lot of features are required, the complexity of the language system must not be excessive. Furthermore, the interaction of the control/rule language subsystem with the rest of the program (written in FORTRAN 77) must be clearly defined and reasonably portable. Functions of the language subsystem must be callable from the core program. Provisions for the transfer of input data and results of various types in both directions must be supplied. The language subsystem needs ways to access variables of the core system in order to read and manipulate molecular data.

3. THE BASICS: FORTH

The full set of requirements and desirable features are not easy to cover. Browsing the concepts of many computer

languages, only a single candidate remained as a starting point for the development of the WODCA system language: FORTH.

FORTH[24] is a computer language which was developed in the early 1970s. Its original application was the control of a radio telescope. In the early days of the home computer era this language had quite a number of devoted followers, but it lost widespread support in later years with the increase of computational power and especially available memory in these machines. Currently, FORTH is experiencing a renaissance as a language of choice for real-time machine control applications. Processors which execute FORTH code directly in hardware have been available for several years. The biggest influence of FORTH felt today in everyday computer work is its status as an ancestor of PostScript.

Standard FORTH (FIG-FORTH, ANSI FORTH) is a powerful language with an extremely compact core and a very small run time system. If the primitives are coded in assembler, the core functionality can be packed into 4K ROM. FORTH is compiled into *threaded code*. Functions or commands (both are called *words*, there is no distinction, and there is no hard difference between intrinsic core words and those added later by compilation) which call each other are connected by pointers after compilation. This means, no lookup of strings or tokens takes place at execution time. The execution engine just chases pointers and is therefore extremely fast. Computational results are accumulated on a stack, which can be manipulated in a large number of ways. Variables and constants are special words which push the address of the value storage area or the binary and preparsed value itself on the stack, respectively. Retrieval and assignment of variable values are done by indirection and store operators. Branches and loops examine and consume stack elements. In contrast to classical programming languages, most operations take place on the stack without intermediate variables. FORTH uses postfix notation; that is, the operator follows the arguments.

Raw FORTH, notwithstanding its elegance, was found to be too dangerous as a control language because it uses stack data directly as pointers and has no knowledge about the data types on the stack. Recovery in the case of errors is not much of a problem on dedicated FORTH systems, which have exception handlers, but crash recovery can be awfully complicated in the case of a concurrent multiprogram system such as WODCA running in a multiuser environment. Additionally, no high-level data types such as strings are part of the standardized elementary data type set, which includes only bytes, integers, and probably floats. They are emulated rather awkwardly by high-level constructs.

The conclusion is that FORTH is basically a very elegant solution to the described problems but requires some addons to be really useful as a program control language in the specific environment. The most important addition deemed necessary was type-tagging of the stack entries. This implies built-in support for many different data types such as floats and strings and their pointers or addresses of executable words. This is a clear contrast to unclassified integers and bytes in standard FORTH. Looking at the type tags, the execution engine refuses to use unsuitable entries as pointers and rejects other potentially hazardous operations. Thus, operation is much more secure and nearly foolproof. Additionally, the built-in commands are aware of the data types of the items they are working with and automatically convert from the presented format to an internal suitable format before the actual computation commences. In typical cases no explicit instruction-driven transformation between integers, floats, strings, and more specialized data types is necessary, although transformation and casting words are implemented for tricky cases. The types of stack entries can be read and set by special words, so new data types can be created. PostScript-type stack marks, which are very handy for array manipulations, were implemented completely as FORTH words. Words which overload standard operators or words are coded easily using the stack entry tags. See Figure 1 for some examples of FORTH code.

Compiled words are stored in dynamically allocated C structures. Refer to Figure 2 for an example. The execution engine, a compiled C subroutine, traverses the code fields of the structures and invokes the proper actions. In the case of function/subroutine calls, the associated data field contains the address of another compiled word and the executor calls itself recursively with this word address. Other elementary operations which correspond to the built-in basic words or tasks like variable readout and storage are performed by hard-coded small C subroutines. Altogether, some 14 stack entry types and 26 code field types are recognized in the standard setup.

Various features found only in FORTH-type languages are especially convenient for function development and debugging. A good example is array limit checks. Functions which use the array indexing operator () contain the address of the operator version valid at compile time. If the operator is redefined later, for example in a way that includes array boundary checking, all words using this operator which are compiled after the redefinition contain the reference to the new instance and perform array boundary checking, while the probably more mature functions defined earlier before the redefinition proceed significantly faster without the checking overhead. Both built-in functions and user-defined words can be redefined an arbitrary number of times. The redefinition process is very useful for local variables, which are words themselves. The words are grouped into vocabularies. Various methods of access control to vocabularies are supported, so that potentially hazardous but indispensable functionality can be hidden and made inaccessible for general purposes. Words carry a number of additional attributes such as error catcher flags, traceability suppressor bits, optimizability, and direct execution precedence at compile time, so that the behavior of a word can be finely tuned.

The complete system was implemented in about 3.300 lines of C and about 1.000 lines of FORTH. This includes FORTH compiler (not to be confused with the cross-compiler introduced in section 4.1, the compiler explained here reads *FORTH* tokens and encodes them for the execution engine), file I/O including redirection and communication with an external graphical debugger, exception handling, and an extensive set of mathematical and string handling functions, altogether 142 words in the C source and some 70 in FORTH start-up files. These numbers are only the basic set and exclude the functions created especially for WODCA. A major part of the system is written in FORTH itself and compiled into the core system at run time from start-up files. This takes only a few seconds on a Sparc I workstation. The interactive command interpreter, the debugging routines, the word disassembler, and many utility functions were implemented in this way. The expansion of the FORTH base system with routines written in the language itself is a powerful and elegant concept. The interactive command line interpreter (only some 10 lines of FORTH code) is even reentrant, which is very useful if code is being debugged and some expressions need to be evaluated in the midst of the debugging session.

```
? 1 2 + .↵
3
```

'?' is the prompt symbol and ↵ a pushed 'return' key. Here first 1, then 2 are pushed on the stack. Add with the '+' operator. The two topmost stack entries are removed and replaced by the result. Print the unformatted result with '.', simultaneously destroying the top stack entry.

```
? : round 0.5 over 0 < if - else + then ->i ;↵
? 1.2 round type space 1.7 round type space -1.7 round .↵
1 2 -2
```

Compile a new word called 'round'. Compilation is between ':' and ';'. 0.5 pushes a floating point number on the stack. 'over' copies the second stack entry from the top onto the top once more again. It is then compared against zero ('0' and '<'). The result is an integer one or zero on top of the stack. 'if' takes this stack entry and branches. The 0.5 value, which is now the top of the stack, is either added or subtracted from the element below. 'then' terminates the if-clause. The single floating point result is finally truncated to an integer with '->i'.

```
? 5 "x" variable↵
? : sqr dup * ;↵
? x @ sqr .↵
25
```

Create an integer variable "x" and initialize it to five. The initializer determines the variable type. Compile a squaring function "sqr". The word 'x' puts the variable value field pointer on the stack. The '@' operator replaces the pointer with the value pointed to. It is squared and printed.

```
? : fac ->i dup 1 <= if drop 1 else dup 1- "fac" ' exec * then ;↵
```

Create a recursive faculty operator. The embedded "fac" puts a string on the stack which is replaced by the operator "'" with the address of the compiled word (which is not yet known at compile time). 'exec' executes at this address.

```
? 5 fac x !↵
```

Compute the faculty of five and store it in variable x. '!' takes a variable address and a value on the stack, stores the value at the address and removes both stack entries.

```
? 5 fac "Result is %d.Bored ? " print query↵
Result is 120. Bored ? yes↵
```

Format a nice reply.

Figure 1. Tour de FORTH in FORTH programming. This scission log exercises some elementary statements. The scope of this paper does not allow an extensive introduction into the concepts of this language.

The interface to the WODCA FORTRAN main program is simple. Basically, only functions to execute a string as a sequence of FORTH commands and to push and pop various FORTRAN data to or from the stack are needed. Furthermore, a function is provided which registers a FORTRAN variable or array (preferably in a COMMON block) as an access word in the FORTH system. Another interface function synchronizes I/O of the FORTH subsystem and the FORTRAN main program in the case of shared I/O channels (standard output, standard input). Finally, a few utilities which enhance the embeddability such as existence checks of FORTH words by FORTRAN routines were added. This small set of functions depend on the C/FORTRAN cross-language calling conventions of the OS. Ports currently exist for mainstream Unix machines and VAXen. These two computer families have drastically different calling conventions, especially as far as FORTRAN strings are concerned.

Because the interface is so small and well-defined, the FORTH subsystem has been successfully transferred to other applications, most notably a programmable report generator for molecular data and a molecular property computation and extraction program for QSAR studies.

4. HIGH LEVEL: VERGIL+++

While the FORTH subsystem works very well and efficiently, some unexpected acceptance problems were encountered when a broader user base was exposed to it. FORTH routines tend to be extremely concise, and the concepts employed in this high-density coding style are not readily grasped by people familiar only with classical programming languages. It seems to be true that considerable experience is required to understand at a glance the processes involved even in short routines. A disciplined programming style is required to avoid the abuse of FORTH as a *write only language*, as it has been nicknamed.

In the laboratory of Prof. Gasteiger/TU Munich another control language was already being used on a regular basis. The language is VERGIL, the rule-encoding language of the EROS6.1 reaction prediction program.[7] This language adheres to the structures of mainstream languages such as FORTRAN and Unix shells. A connection between VERGIL and the FORTH system was therefore conceived as desirable. However, the language specification of basic VERGIL as found in EROS is not sufficient for the needs of WODCA. Some

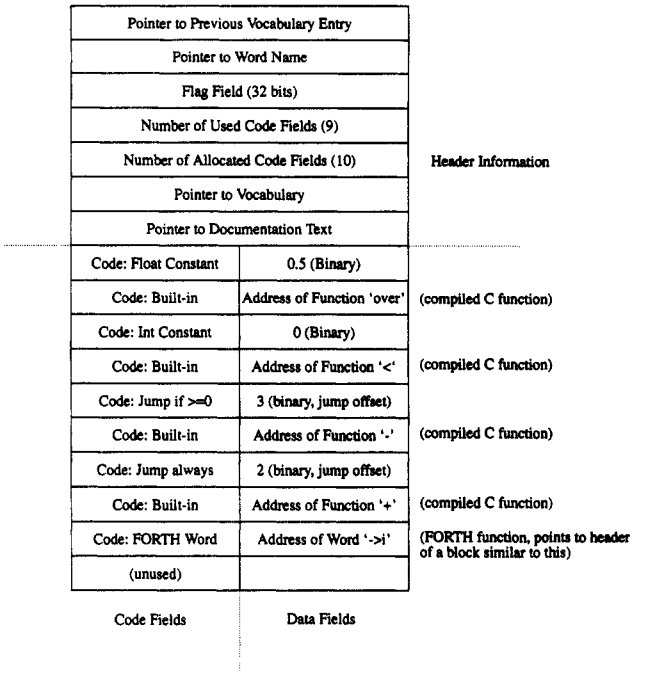

Figure 2. Structure of the FORTH word *round* from Figure 1 after compilation in core memory. Every box is a 4-byte memory word. During compilation, the word structure is enlarged in blocks for efficiency reasons, so there may be some unused word slots left. This structure occupies some 120 bytes including the word name, which is a negligible quantity. The number of words which can be held in core memory is large enough to pose no limit for practical applications.

enhancements such as a string variable and expression types, static local variables, true location-independent subroutines, local arrays, and various looping constructs were added to the core definition. In contrast to basic VERGIL, recursive and cross-recursive function calls are allowed.[25] The enhanced but upward-compatible dialect was named VERGIL+++, but the name does not imply that it is object-oriented in any fashion.

4.1. Cross-Compiler. VERGIL+++ source is translated into the WODCA FORTH dialect by a cross-compiler. This proved to be surprisingly easy, because the sequence of parse tree tokens analyzed by a LR(*n*) parser, such as *yacc* or *bison*, corresponds neatly to an executable FORTH token sequence. Renaming the functions and constructs of VERGIL to the FORTH standards is a trivial operation. Sometimes a few stack element swap operations need to be inserted, but otherwise there is an exceedingly easily exploited relationship present. The translation process proceeds *directly* without intermediate storage of tokens in a parse tree.[26] The necessary stack element reordering operations can be inserted automatically in the output sequence, following a purely mechanical scheme. See Figure 3 for an example of automatically translated code. The purpose of the code in Figure 3 is explained further in paragraph 5.

The cross-compiler was implemented with the standard Unix tools *yacc* and *flex*.[27] The grammar of VERGIL is not difficult to parse. It is context free and has reserved keywords which must not be used as variable or function names, and a scanner with a one-character look-ahead is sufficient for token recognition. While a stateless scanner could be employed easily, it was found to be convenient to use a two-state implementation to treat the negation/minus-operator differentiation problem. Comments, white space with the exception of line terminators,[28] and line-continuation markers are suppressed in the lexer. The *yacc*-generated parser uses error productions to catch common errors and tries otherwise to skip to the next end-of-line token (classical *panic recovery*). Error reports are clear-text with line-numbers. Offending lines are highlighted in the editor window of the cross-compiler.

Arithmetic priorities and semantic analysis are handled automatically by the parser and need not be programmed at all. Variable and expression type conversions are likewise unnecessary, because the tagged stack extension in the FORTH dialect of WODCA will handle this automatically at execution time. The generated FORTH code is unoptimized and generally does not follow the lines that an experienced programmer would choose if the routine was programmed directly in FORTH,[29] but the code still works reasonably fast after compilation. Because special line marker commands are inserted into the generated FORTH code if such an option is set in the cross-compiler, debugging of the VERGIL code in a line-oriented fashion is possible even after compilation.

Because VERGIL does not support the full set of operations possible with the FORTH subsystem and the automatically generated code is notably slower, the FORTH interface is still accessible and used for computationally complex or tricky procedures, even if everyday development of rating and filtering functions for the synthesis-planning program is now largely carried through in VERGIL. Since the FORTH code is not necessarily submitted immediately to the FORTH system, manual optimization of the machine-generated code is possible.

The cross-compiler and a graphical user interface for the debugger are not integrated into the core WODCA system but are run as a separate program. See Figure 4 for a snapshot. Modules can be precompiled into FORTH code in a separate editing session, or the interactive development option is chosen. This means that the cross-compiler is run concurrently with the core WODCA system. In this case bidirectional data exchange proceeds in real time through dedicated communication channels between those two programs.[30] Users which intend only to use precompiled standard functions in the synthesis planner and do not develop new functions or chase bugs in older ones do not need to start the debugger/cross-compiler program. They use less memory and are not confused by functionality that they never employ.

4.2. Reaction Rule File Preprocessor. Once the link between the rule language of EROS6.1 and the command language of WODCA has been established, the use of WODCA as an interactive test facility for EROS reaction rules comes natural. The rule definition files of EROS6.1 contain the rule source code for reactivity/rate prediction on the reactant ensemble side of a reaction and heat-of-reaction/merit/turnover evaluation on the product side buried deep in auxiliary reaction scope definitions. However, the core rule content can be extracted by a preprocessor, which has been implemented as another part of the debugger/cross-compiler system. These extracted functions are augmented with compatibility heaters,[31] translated, and sent to the WODCA core. WODCA can be be used as an interactive test environment for those rules. Within WODCA molecules are retrieved or created, possible reaction centers are marked, and the rule results on the reactant and product sides as well as the reaction generators (which cut and make bonds in the reaction center) can be tested in detail with problematic molecules, so the reasons for a specific or suspicious behavior of a rule become obvious very fast under tightly controlled circumstances, when every step of the evaluation process is monitored with the battery of debugging tools. Reaction functions which were modified during debugging and extracted from the rule files can be merged back automatically into the

```
# strategic bonds: all substituents at aromatic systems
void function sb_all_arosub()
def bd: integer
loop bd=1,NBOND
do
 if B1AROMATIC(bd) and not BAROMATIC(bd) then
    if not AAROMATIC(BONDATOM1(bd)) then
       if ATOMICID(BONDATOM1(bd)) <> HYDROGEN then
          SBATOM1(bd) = CHLORINE # simple pseudo-appendix atoms
          SBATOM2(bd) = HYDROGEN
          SBCUTMODE(bd) = CUT_THROUGH # no fancy complex rearrangements
          SBRATING(bd) = 100 # this is GOOD!
       endif
    endif
 endif
done
endfunc
```

```
: sb_all_arosub ( --- 0 parameters --- )
 0 "bd" variable
 1 "NBOND" ' exec @ swap do
    i "bd" ' exec !
    "bd" ' exec @ B1AROMATIC swap () @ "bd" ' exec @ BAROMATIC swap
    () @ not and if
       "bd" ' exec @ BONDATOM1 swap () @ AAROMATIC swap () @
       not if
          "bd" ' exec @ BONDATOM1 swap () @ ATOMICID swap () @
          "HYDROGEN" ' exec <> if
             "bd" ' exec @ CHLORINE swap SBATOM1 swap () !
             "bd" ' exec @ HYDROGEN swap SBATOM2 swap () !
             "bd" ' exec @ 3 swap SBCUTMODE swap () !
             "bd" ' exec @ 100 swap SBRATING swap () !
          then
       then
    then
 loop
"bd" forget
;
```

Figure 3. Primitive example of a strategic bond detection rule of the WODCA system and the produced FORTH cross-compiler code. The function sets all bonds to substituents at aromatic systems strategic, does not introduce any complex treatment of the open valences, and assigns a flat merit value. This code example does not contain any debug information for later stepping along the lines of the original VERGIL source because no such option was selected in the cross-compiler. Compiler flags were set to allow for true local variables and recursive calls, which makes the generated code slower and more complex, but the function works under all circumstances.

original file. Figure 5 summarizes the information flow in the WODCA/EROS environment.

5. EXAMPLE FUNCTIONS

Some short and abbreviated examples for functions employed in the synthesis planner part of WODCA might be helpful for further understanding.

Functions generally have access to a bundle of external arrays which hold a basic set of physicochemical and topological properties of the ensemble, individual molecules, atoms, bonds, and rings. These arrays are filled by the core program for the current ensemble. Additional derived properties (which can be declared persistent or local) can be set up and computed in functions. The basic property set made available by WODCA is a superset of these properties used in the EROS reaction prediction program series. Furthermore, some predefined output arrays for strategic bond detection are used here.

The function listed in Figure 3 is simple but illustrates some basic points. This type of function is used to mark bonds as strategic for synthesis planning. This example adds to the list of strategic bonds in the molecular ensemble under examination those bonds which link a nonaromatic substituent other than hydrogen to the aromatic core. The function starts with a loop on all bonds. Bonds which are themselves nonaromatic (BAROMATIC) but are α to an aromatic bond (B1AROMATIC) are selected. In the data model of WODCA and EROS, bonds are registered in both directions because among the set of supplied bond properties some are directional. Therefore, the next lines of the function select only bonds leading to the substituent (which is nonaromatic, *not* AAROMATIC). Once those bonds have been filtered for directionality, the final check is whether the substituent is a hydrogen atom or not. If not, simple pseudoatoms which will be added if the bond is cut in a retrosynthetic direction, are defined (hydrogen for the aromatic core, chlorine for the substituent) and the rating field (SBRATING) is set to a high value. If the bond is cut, it is just severed (SBCUTMODE).[32]

More realistic functions like the one partially visible in the hardcopy of the cross-compiler tool (Figure 4) are too long (typically several hundred lines) to be explained in detail in this paper. The function in Figure 4 consists of several independent subroutines, and more than one such function set

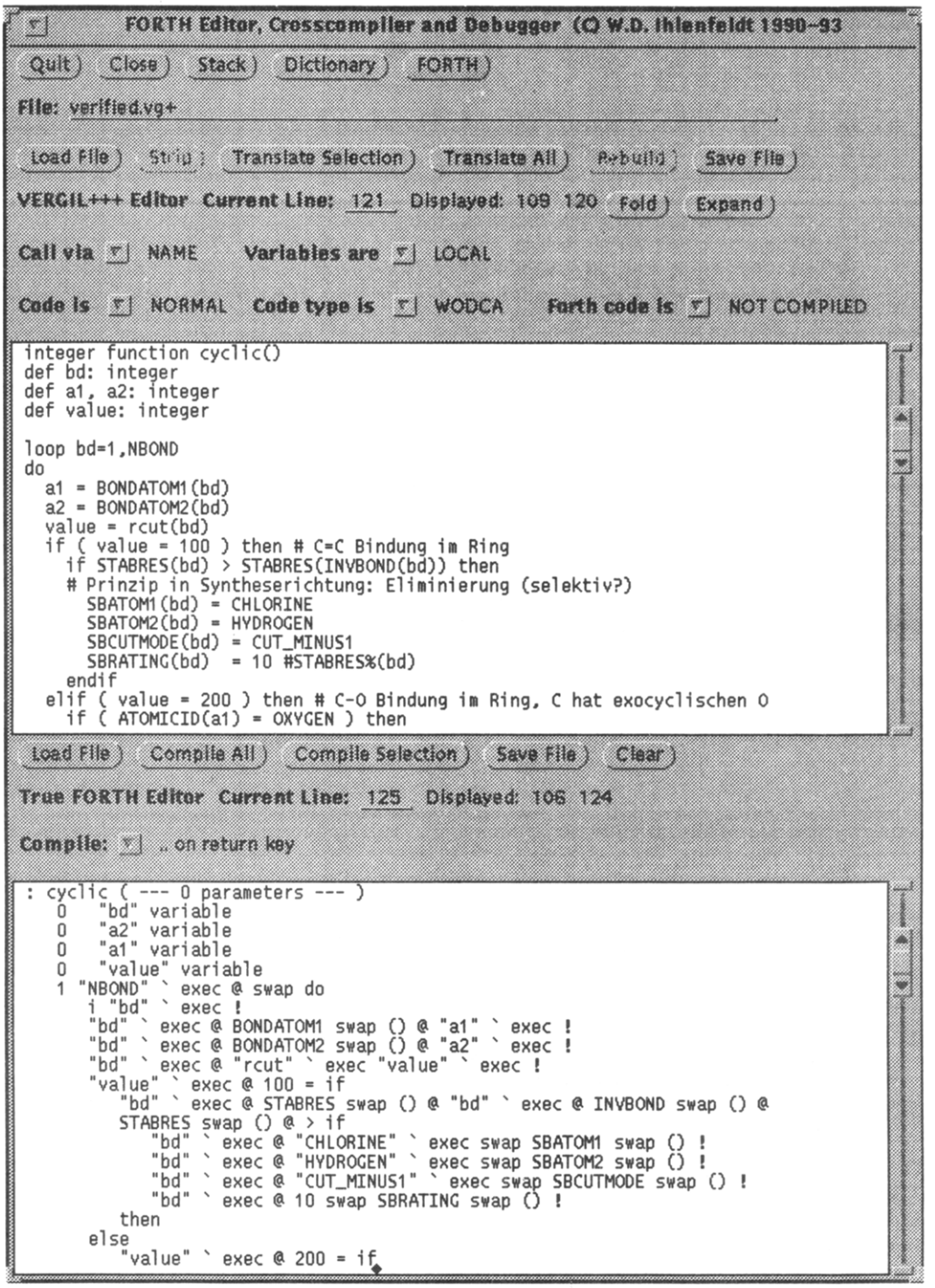

Figure 4. The interactive cross-compiler. The upper text window shows some of the VERGIL+++ input code, the text window below contains FORTH source output generated by the cross-compiler. This code either is sent directly to the running WODCA core system via a communication channel or can be edited, stored, etc., for future (re)use. This graphical tool also serves as a debugger for runing FORTH functions in the WODCA core. Code can be stepped through on the VERGIL or FORTH level, breakpoints may be set, the FORTH stack can be examined, variable values are accessible, etc. Basically all functions of a classical debugger are available. This set of interactive features make the development and maintenance of functions much easier than the classical compile/link/print/curse/edit circuit.

may be applied sequentially to detect strategic bonds. The actually employed functions in the WODCA system use a plethora of more advanced features for strategic bond designation. A differentiated merit value is computed from the physicochemical and topological characteristics of the bond examined. More complicated cut modes, which may result in the elimination of α, β, or δ atoms or groups, may be requested by the function. Strategic bonds can be coupled for synchronous reaction or to model parallel identical or similar reactions at different centers. The atoms which are added to the open valences should be chosen with care to maximize the chance of selective attack, or there might be no such atoms at all. In this case an internal recombination or elimination might take place, and so on. Functions are also designed to cooperate, so a mediocre rating found with a specific function will not overwrite the definition found by another function which explores other possible retrosynthetic reaction classes. However, the basic form of those functions is similar.

Functions which rate the fit of a potential starting material for a compound under examination are a little bit different from the strategic bond detection family. While the latter scribble directly into external variables, the match functions

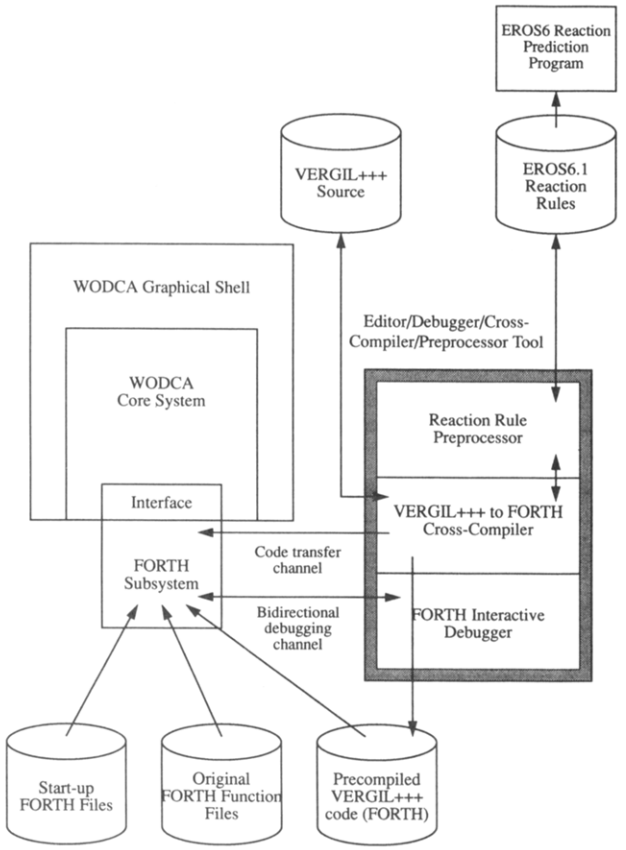

Figure 5. Information flow in the WODCA and EROS environment. Only the transport paths of FORTH source, VERGIL source, EROS6.1 reaction rule code, and debugging information are shown. Additionally, there are information pathways within the WODCA system and on the graphical level between the WODCA user interface shell and the peripheral programs.

are presented with series of specific overlaps, which are typically close to a maximum common substructure[33] or sometimes identical. The stereochemical and topological fit and the (selective) reactivity of the bonds which are to be broken are examined. In order to judge the fit, the functions are allowed access to the properties of both synthesis target and prospective precursor and to the overlap vector. The functions return a numerical function value for this overlap and write comments to a log. These functions tend to be quite complex because they try to analyze factors such as reactivity, chances for selective attack, and possible chiral induction in the case of missing stereo centers. Because of the sheer number of possible reasonable overlaps, these are not stored in some continuous order in memory similar to the potential strategic bonds but are mostly discarded immediately after generation if they are not competitive with prior good matches. Compared to strategic bond detection routines, efficiency of execution is much more an issue here because of the number of evaluations performed. If a nontrivial pool of starting materials is checked against a complex target, several hundred or thousand overlaps are presented to those functions.

6. SUMMARY AND CONCLUSION

For the control of a complex synthesis-planning program and for the encoding of the knowledge used by this program a pair of two control and command languages was designed and implemented. The pair covers the requirements for efficiency, completeness, implementation effort, comprehensibility, usability, and debuggability in a convincing manner. The combination of this comprehensive set of features into a single language would have been very difficult and certainly more time-consuming. By the introduction of a link between the rule language of the EROS6.1 reaction prediction system and the control language of the WODCA synthesis planar, synergistic effects were achieved. WODCA uses the command and rule language pair successfully for a wide variety of tasks including strategic bond detection, structure filtering, precursor mapping merit evaluation functions, and batch commands. Furthermore, with the language link WODCA can be conveniently used as an interactive test bench for the reaction rules of EROS6.1.

7. PERSPECTIVES

Since the power of computers is ever increasing, the impact of the overhead of interpreted control and rule languages becomes continuously less severe. In recent times the need for powerful general-purpose portable tool command languages has been recognized by the computer language community. In the last years a number of implementations of such languages have become available. These languages are carefully designed to become an embedded integral part of programs. They are typically interpretive and string-based, but may feature also a limited amount of preprocessing (tokenization or parse tree construction, even with constant expression folding) in order to speed up execution.

The most notable of these languages is Tcl (tool command language) by J. Ousterhout at the University of Berkeley.[34,35] Other less complete and not as widely available languages of this type are PYTHON, ICI, SLANG, the upcoming PERL 5,[36] and various LISP dialects. Because Tcl is so easily expandable, a wide selection of add-ons have appeared including a complete X11 interface complete with an interactive interface builder (Tk and XF), mathematical and Unix OS access functions (TclX), database interfaces (among others to the POSTGRES and SYBASE database systems), and a large number of smaller utilities. Hopefully this movement toward generally applicable program control languages will lead to some standardization. Each of the languages mentioned has more features and is better tested than any specialized language which is implemented only to control a single program. All of them are designed to easily integrate extensions which are necessary to access and manipulate the application-specific data structures. The general acceptance of a standardized core control language with a limited set of application-specific extensions would drastically reduce the time and effort required until one can use a new powerful program skillfully, because it avoids the need to master another limited, weird and quirky custom syntax beforehand. For these reasons, the CACTVS chemical data management system[37] currently under development has been designed from the beginning around Tcl as the control and command language. The initial experience certainly indicates that this was a justified and overdue move toward compatibility and usability.

ACKNOWLEDGMENT

I gratefully acknowledge financial support from Studienstiftung des deutschen Volkes (scholarship) and Deutsche Forschungsgemeinschaft (DFG, research grant for the continued development of the WODCA system). The presented work was carried through in the laboratory of Prof. Gasteiger, TU Munich, and I am indebted for many helpful discussions with Prof. Gasteiger and the members of his group.

REFERENCES AND NOTES

(1) Current address: Toyohashi University of Technology, Dept. of Knowledge-based Information Engineering, Tempaku, Toyohashi 441, Japan.

(2) Gasteiger, J.; Hutchings, M. G.; Christoph, B.; Gann, L.; Hiller, C.; Löw, P.; Marsili, M.; Saller, H.; Yuki, K. A New Treatment of Chemical Reactivity: Development of EROS, an Expert System for Reaction Prediction and Synthesis Design. *Top. Curr. Chem.* **1987**, *137*, 20–73.

(3) Jorgensen, W. L.; Laird, E. R.; Gushurst, A. J.; Fleischer, J. M.; Gothe, S. A.; Helson, H. E.; Paderes, G. D.; Sinclair, S. CAMEO: A Program for the Logical Prediction of the Products of Organic Reactions. *Pure Appl. Chem.* **1990**, *62*, 1921–1932.

(4) Hanessian, S.; Franco, J.; Gagnon, G.; Laramée, D.; Larouche, B. Computer-Assisted Analysis and Perception of Stereochemical Features in Organic Molecules Using the Chiron Program. *J. Chem. Inf. Comput. Sci.* **1990**, *30*, 413–425.

(5) Smith, G. S.; Woodruff, H. B. Development of a Computer Language and Compiler for Expressing the Rules of Infrared Spectral Interpretation. *J. Chem. Inf. Comput. Sci.* **1984**, *24*, 33–39.

(6) Hanessian, S.; Franco, J.; Larouche, B. The Psychobiological Basis of Heuristic Synthesis Planning–Man, Machine and the Chiron Approach. *Pure Appl. Chem.* **1990**, *62*, 1887–1910.

(7) Röse, P.; Gasteiger, J. EROS 6.0. A Knowledge-Based System for Reaction Prediction–Application to the Regioselectivity of the Diels-Alder Reaction. In *Software Development in Chemistry 4*; Gasteiger, J., Ed.; Springer: Heidelberg, 1989.

(8) Gasteiger, J.; Ihlenfeldt, W. D.; Röse, P.; Wanke, R. Computer-assisted Reaction Prediction and Synthesis Design. *Anal. Chim. Acta* **1990**, *235*, 65–75.

(9) Johnson, A. P.; Marshall, C. Starting Material Oriented Retrosynthetic Analysis in the LHASA Program. 3. Heuristic Estimation of Synthetic Proximity. *J. Chem. Inf. Comput. Sci.* **1992**, *32*, 426–429.

(10) Simon, V. Ph.D. Thesis, Technical University, Munich, 1993.

(11) Corey, E. J.; Wipke, W. T.; Cramer, R. D.; Howe, J. W. Computer-Assisted Synthetic Analysis:Facile Man-Machine Communication of Chemical Structur by Interactive Computer Graphics. *J. Am. Chem. Soc.* **1972**, *94*, 421–430.

(12) Corey, E. J.; Wipke, W. T.; Cramer, R. D.; Howe, W. J. Technique of Perception by a Computer of Synthetically Significant Structural Features in Complex Molecules. *J. Am. Chem. Soc.* **1972**, *94*, 431–439.

(13) Corey, E. J.; Cramer, R. D.; Howe, W. J. Computer-Assisted Synthetic Analysis for Complex Molecules–Methods and Procedures for Machine Generation of Synthetic Intermediates. *J. Am. Chem. Soc.* **1972**, *94*, 440–459.

(14) Wipke, W. T.; Ouchi, G. I.; Krishnan, S. Simulation and Evaluation of Chemical Synthesis–SECS. An Application of Artificial Intelligence Techniques. *Artif. Intell.* **1978**, *11*, 173–193.

(15) Yanaka, M.; Nakamura, K.; Kurumizawa, A.; Wipke, W. T. Automatic Knowledge Base Building for the Organic Synthesis Design Program (SECS). *Tetrahedron Comput. Methodol.* **1990**, *3*, 359–375.

(16) Loftus, F. G. An Overview of Synthesis Planning Software. In *Chemical Information Systems–Beyond the Structure Diagram*; Bawden, D., Mitchell, E. M., Eds.; Ellis Horwood: Chichester, U.K., 1990; pp 92–104.

(17) Corey, E. J.; Long, A. K.; Mulzer, J.; Orf, H. W.; Johnson, P.; Hewitt, A. P. W. Computer-Assisted Synthetic Analysis. Long-Range Search Procedures for Antithetic Simplification of Complex Targets by Applications of the Halolactonization Transform. *J. Chem. Inf. Comput. Sci.* **1980**, *20*, 221–230.

(18) Shortcliffe, E. H. *Computer-based Medical Consultations: MYCIN*; Elsevier: New York, 1976.

(19) Lindsay, R. K.; Buchanan, B. G.; Feigenbaum, E. A. *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL project*; McGraw-Hill: New York, 1980.

(20) Gasteiger, J.; Ihlenfeldt, W. D. The WODCA System. In *Software Development in Chemistry 4*; Gasteiger, J., Ed.; Springer: Berlin, 1990; pp 57–65.

(21) Ihlenfeldt, W. D.; Gasteiger, J. Synthesis Planning based upon the Similarity of Chemical Structures. In *Software Development in Chemistry 5*; Gmehling, J., Ed.; Springer: Berlin, 1991; pp 59–67.

(22) Gasteiger, J.; Ihlenfeldt, W. D.; Röse, P. A collection of computer methods for synthesis design and reaction prediction. *Recl. Trav. Chim. Pays-Bas* **1992**, *111*, 270–290.

(23) Gasteiger, J.; Ihlenfeldt, W. D.; Fick, R.; Rose, J. R. Similarity Concepts for the Planning of Organic Reactions and Syntheses. *J. Chem. Inf. Comput. Sci.* **1992**, *32*, 700–712.

(24) Knecht, K. *Introduction to FORTH*. H. W. Sams & Co.: Indianapolis, 1982, or any other introductory textbook.

(25) A complete syntax graph of VERGIL and VERGIL+++ is available on request.

(26) Actually, in the interest of user-friendliness, the source code is parsed twice. In the first pass, only error messages are output. The code generation is suppressed. The second pass, which is identical to the first, is initiated only if no errors were found. The translation process is so fast that buffering of produced code in some temporary storage was deemed unnecessary.

(27) Due to errors in the Sun version of standard *lex*, the extended Gnu FSF version called *flex* was used. Commands (but not variables) in VERGIL are case-insensitive. *flex* is also more convenient than *lex* because it can directly generate case-insensitive scanners without the need for large character folding expressions, and the generated lexer is definitely faster. Standard *yacc* worked well enough, so no attempt was made to replace it by the FSF extended equivalent *bison*.

(28) VERGIL is not a completely form-free language. Each statement occupies one line, if no line-continuation symbol (backlash in the last used column) is added, similar to Unix shell languages.

(29) The main problem with the automatically generated code is that comparatively inefficient temporary variables are used instead of stack manipulations. These variables have to looked up by name, because they might be locally overlaid by recursive function calls. However, this is the style found in the original VERGIL code, and an automatic optimization toward efficient stack use and removal of superfluous temporary variables is a formidable problem. Another reason for inefficiency are loops, which is FORTH are implemented as stack operations without real loop variables. However, a local pseudoloop variable has to be created and continuously updated if VERGIL loops are cross-compiled. As a rule of thumb, directly hand-coded FORTH routines are 60% shorter and 2 or 3 times as fast as cross-compiled code.

(30) Actually, there are three programs involved: the WODCA core system, its graphical user interface (this is a separate program), and the cross-compiler, which all exchange messages. There are typically quite a number of additional programs which surround WODCA (displays, etc.) and are in parallel, but they do not take part in this particular communication subsystem.

(31) These headers essentially adjust the scope of array indexes. Reaction rules in EROS index the atoms in the perceived reaction center, while WODCA uses indexes which span the whole molecular ensemble. The compatibility headers declare the global arrays which hold the physicochemical properties EROS uses to compute reactivities as reaction center oriented, which in turn instructs the cross-compiler to insert a mapping function call in the indexing expression if such an array is accessed. The mapping function projects the reaction center atom index to the absolute index.

(32) This function fails if the substitutent is aromatic itself, so the central bond in biphenyl will not be marked.

(33) WODCA does not use a traditional MCSS algorithm. The algorithm, which will be published separately, is also distinctly different from those used in CHIRON and LHASA. The MCSS is among the generated overlaps only if there are potentially reactive anchor points between the MCSS and the starting material.

(34) Ousterhout, J. Tcl: An Embeddable Command Language. *Proc. USENIX Winter Conference*; 1990; pp 133–146.

(35) Ousterhout, J. K. *Tcl. and the Tk Toolkit*; Addison-Wesley: Reading, MA, 1994.

(36) PERL must not be confused with PEARL. These languages are completely different.

(37) Ihlenfeldt, W. D.; Takahashi, Y., Abe, H.; Sasaki, S. Computation and Management of Chemical Properties in the CACTVS System: An Extensible Networked Approach towards Modularity and Compatibility. *J. Chem. Inf. Comput. Sci.* **1994**, *34*, 109–116.