PRESENTATION OF SOFTWARE DEVELOPMENT INFORMATION IN K2*

HAUSI A. MÜLLER, DANIEL M. HOFFMAN,
R. NIGEL HORSPOOL, MICHAEL R. LEVY

Department of Computer Science, University of Victoria, Victoria, BC. V8W 2Y2.

ABSTRACT

K2 is a new approach to solving two of the most difficult problems in the area of programming-in-the-large — mastery of the structural complexity of large software systems and effective presentation of the information accumulated during the development concerned with the specification, design, integration, and maintenance of software systems at a higher level than that of a single module. The major objective of K2 is to effectively represent and manipulate the building blocks of a software system and their myriad dependencies, thereby aiding the development phases of the project.

The K2 environment is an integrated collection of tools that allows the members of a complex software project to organize development information by manipulating system description documents of various types and by establishing links between those documents. System description documents support the project from the early specification and design phases through the long-term maintenance phases. The K2 system will improve the productivity of the designer, programmer, integrator, and maintainer.

Keywords : Programming-in-the-large, software development environment, programming-in-the-many, software specification, compiler generators, version control, maintenance, software life cycle, K2, Rigi, Autodoc, Mk*.

RÉSUMÉ

K2 est une approche vers la solution de deux problèmes difficiles rencontrés en programmant de grands systèmes logiciels: l'ordre de la complexité structurale et la présentation de l'information accumulée en développant un système. La programmation en gros concerne la spécification, l'analyse, l'intégration et l'entretien à un niveau plus élevé qu'un module. Le but de K2 est la représentation et la manipulation de blocs logiciels et de leurs connexions.

L'environnement K2 est un assemblage intégré des outils pour l'organisation de l'information concernant la développement d'un système. Cet environnement permet à un programmeur de manipuler les fichiers de description du système et de créer des liens entre ces fichiers. Chaque membre d'un projet utilise ces fichiers comme un moyen de communiquer et coordonner. K2 améliora la productivité pendant la conception, la programmation, l'intégration et l'entretien d'un système logiciel.

Mots-clés : Programmation des grands systèmes logiciels, environnement de développement logiciel, spécification de logiciel, générateur de compilateurs, contrôle des révisions, entretien de logiciel, cycle de vie logicielle, K2, Rigi, Autodoc, Mk*.

1. INTRODUCTION

A software development environment deals with three distinct problem areas — *programming-in-the-small, programming-in-the-large,* and *programming-in-the-many* DeRemer and Kron (1976); Haberman and Notkin (1986).

Programming-in-the-small refers to those aspects of software development that are dealt with in *programming environments.* It involves the development (construction, analysis, compilation, execution, optimization, debugging, testing, and monitoring) of a *single* module of a software system (*i.e.*, the representation and manipulation of abstract syntax trees and code fragments).

* Recd. May 1988, Accepted July 1988.

Programming-in-the-large refers to those aspects of a software development environment concerned with the specification, design, integration, and maintenance of software systems at a higher level than that of a single module. It includes the organization and representation of system structure, module decomposition, component dependence analysis, interface control, separate compilation, subsystem and composition identification, as well as version and release control, Müller (1986).

Programming-in-the-many refers to those aspects of the software development process concerned with *multiple programmers* working on a project and/or a *large time frame* for a project. It includes issues such as access control, mutual exclusion capabilities, documentation, change logs, network access, and project management, Notkin, *et al.* (1985).

The K2 project concentrates on the programming-in-the-large aspects of software development environments. However, it also addresses many issues in the areas of programming-in-the-small and programming-in-the-many. In particular, a viable implementation of K2 will necessarily deal with all three problem areas.

2. THE PROBLEM

The major objective of K2 is to solve two of the most intricate problems in the area of programming-in-the-large — the mastery of the structural complexity of large software systems and the effective presentation of the information accumulated during the development process. These problem areas include readability and ease of understanding of system descriptions, definition and manipulation of system structures, analysis of component dependencies, interface consistency, integration, mechanisms, composition identification, and version control, Müller and Klashinsky (1988).

Readability and ease of understanding of system descriptions. One inherent difficulty in reading programs is the complexity of the systems they implement, Goldberg (1987). A newcomer should be able to read system descriptions in the same way as an *electronic atlas.* Initially, the *big picture* of the system is presented. Details of the components and their dependencies can be inspected by "zooming in" on sections of this big picture. The level of detail can easily be varied without being overwhelming. The user can swiftly navigate through the myriad system elements and readily identify programs, packages, integration mechanisms, interfaces, specifications, and version families.

Defining system structure. The identification of system components and their dependencies during the design phase of a software system is difficult and crucial. Abstraction mechanisms, modularity, information hiding, hierarchies, integration mechanisms, and reusability have emerged as the most important issues in the definition of software systems. Reusability issues are of especial importance in the development of large software systems. Reinventing the wheel is costly; reuse of development knowledge improves both time and space efficiency as well as the reliability of the overall system.

Interface consistency and integration mechanisms. Interface consistency — including parameter and type checking — involves *intra-module* as well as *inter-module* consistency, Clark (1985); Cooper *et al.* (1986); Hood *et al.* (1986); Lichtman (1986); Tichy (1986). Intra-module consistency is concerned with resource relationships between the syntactic definition part of a module and its implementation parts, whereas inter-module consistency involves relationships between the resources of a component and its environment. It is important that these checks can be performed from the earliest stages of the specification and design phases through the long-term maintenance phases. The checking algorithms must therefore be able to deal with incomplete interfaces. During the development as well as maintenance phases of a software system, some of the following questions may arise: What components need to be recompiled due to an editing change in a particular component? Is a set of modules complete and consistent? What components form a program?

Version and release control. A large, evolving software system may consist of thousands of modules, hundreds of programs, extensive documentation, and large amounts of test data. All large and widely-used systems are subject to a stream of corrections, improvements, customizations, enhancements, and diversifications, Tichy (1979); Leblang *et al.* (1985). Each component of the system, as well as the system itself, exists, therefore, in the form of a collection of versions.

3. THE K2 APPROACH

The K2 environment is an integrated collection of tools that allows the members of a complex software project to organize development information by manipulating *system description documents* of various types and by establishing *links* between those documents. System description documents contain information threads ranging from plain text, to graphs, to digitized pictures. The documents and the links are edited and maintained using a *graph editor* and stored in an underlying *project data base.*

The project data base is a repository for the contents of the system description documents. The *Rigi model* a conceptual network data model, Brodie *et al.* (1984), defines the entities and the structure of the data base. The project members browse and update the building blocks of software systems stored in the project data base using the *Rigi editor.* The top level of the editor presents dependencies among system description documents as graphs in windows on high resolution display workstations. Depending on the document type, the editor runs tools such as structure-oriented editors, browsers, translators, or drawing programs. The user interface of this graph editor is extremely important for the success of the K2 project. The graph editor relies on advanced imaging models, high quality graphics, and uniform screen management. With the advent of powerful personal workstations equipped with pointing devices, high resolution bitmap displays, and sophisticated graphics software, graphs can be drawn conveniently and efficiently, Reiss (1984).

In addition to the system description document editor, K2 will include a set of document compilers for producing typeset project documentation. These compilers trace the component dependencies by extracting information from the project data base. For example, the maintainer may elect to print all information pertinent to a particular module ranging from the specification, to the source code, and to the version history. The integrator may request a drawing of the call graph and the module dependencies of a program. The technical manager can produce typeset documentation of various levels of detail. The design of the document compilers is strongly influenced by Knuth's Web system, Bentley and Knuth (1986); Bentley *et al.* (1986). The goal of the Web programming system is to produce programs that are works of literature; the objective of K2 is to produce system description documents for large, integrated, evolving software systems that are works of art.

The system description documents as produced by K2 are designed to support software projects from the early specification and design phases through the long-term maintenance phases. The documents are intended to be used as a common thread and as a medium for communication and coordination by all members of the project. The primary goal of the K2 system is to attain a better and faster understanding of large, integrated, evolving software systems and, hence, to improve the productivity of the designers, programmers, integrators, and maintainers. Higher productivity, in turn, reduces the costs of large software projects.

The following sections concentrate on the implementation of K2 which has evolved into four major subprojects — *Rigi model, Rigi editor, Autodoc,* and *Mk**. The Rigi model captures the programming-in-the-large aspects of the K2 software development environment. The Rigi editor is a tool to assist the designers, programmers, integrators, and maintainers in defining, exploring, and manipulating the structure of large software systems. Autodoc is a structure-oriented editor for the construction of specification documents. Mk* is a set of tools to facilitate ease of translation among the many K2 system description documents and construction of the K2 document compilers.

4. RIGI

The building blocks out of which systems are built are not at the level of programming language constructs. They are "subsystems" or "packages", each of which is an integrated collection of data structures, programs and protocols.

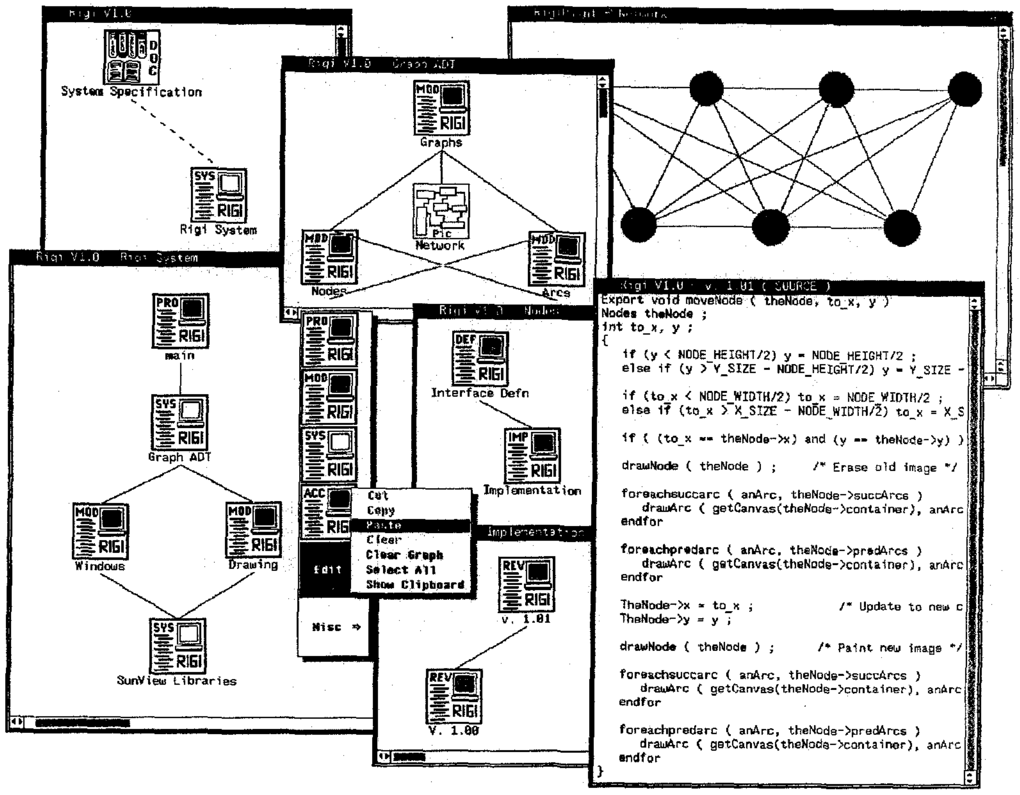
—Terry Winograd, Beyond Programming Languages, Winograd (1979).

4.1 The Rigi Model

The Rigi model is a graph model for the representation and organization of the "bricks" and "mortar" of large, integrated, evolving software systems, Müller and Klashinsky (1988). The nodes and arcs of the graph represent the components of a software system and their dependencies. The abstraction mechanisms *classification, generalization/specialization, aggregation,* and *set* are used to structure the graphs. These abstraction techniques can be applied recursively to form hierarchies. The Rigi model defines disjoint object and dependency classes to model the different types of system components and dependencies. Object classes model entities of building blocks such as *specifications, subsystems, interfaces, implementations, variants,* or *revisions.* The model includes three major classes of dependencies. The *structural* and *change* relationships are induced by the requirements and the provisions of the components, respectively. For example, structural dependencies are used to identify the set of modules that form a program or the web of information threads that document a reusable component. Change dependencies are used to determine the set of direct and indirect clients of a module. The third class, the semantic dependencies, allows the designers of software systems to model any additional relationships that exist among system components.

4.2 The Rigi Editor

The Rigi editor is a tool to edit, maintain, and explore the system descriptions stored in the data base. As in a hypertext editor, Conklin (1987), icons and windows represent closed and open documents, respectively. Initially the editor presents itself as a graph editor. The nodes and arcs of the graph are represented graphically by icons and vectors connecting the icons. The icon types correspond to the object or component classes defined in the Rigi model. Icon and arc menus are used to insert nodes and arcs. Documents may be nested to build aggregation and generalization hierarchies. Entire subgraphs and hierarchies of subgraphs may be duplicated and deleted using the operations *Cut, Copy, Paste,* and *Clear.* Depending on the document type, the editor runs tools such as structure-oriented editors, browsers, translators, or drawing programs which operate on the contents of open system description documents. Figure 1 below depicts a hierarchy of graphs representing a subset of the implementation of the Rigi editor, the background menu of the graph editor, an instance of the RigiPaint program exhibiting a network, and a source module, written in the programming language C, which implements the dragging of nodes in the graph editor.



```
Export void moveNode ( theNode, to_x, y )
Nodes theNode ;
int to_x, y ;
{
   if (y < NODE_HEIGHT/2) y = NODE_HEIGHT/2 ;
   else if (y > Y_SIZE - NODE_HEIGHT/2) y = Y_SIZE -

   if (to_x < NODE_WIDTH/2) to_x = NODE_WIDTH/2 ;
   else if (to_x > X_SIZE - NODE_WIDTH/2) to_x = X_S

   if ( (to_x == theNode->x) and (y == theNode->y) )

   drawNode ( theNode ) ;      /* Erase old image */

   foreachsuccarc ( anArc, theNode->succArcs )
      drawArc ( getCanvas(theNode->container), anArc
   endfor

   foreachpredarc ( anArc, theNode->predArcs )
      drawArc ( getCanvas(theNode->container), anArc
   endfor

   TheNode->x = to_x ;              /* Update to new c
   TheNode->y = y ;

   drawNode ( theNode ) ;      /* Paint new image */

   foreachsuccarc ( anArc, theNode->succArcs )
      drawArc ( getCanvas(theNode->container), anArc
   endfor

   foreachpredarc ( anArc, theNode->predArcs )
      drawArc ( getCanvas(theNode->container), anArc
   endfor
}
```

Figure 1. System description documents

The graph editor offers three principal ways to navigate swiftly through the myriad of dependencies.

Fixed-scale map

Viewing of a fixed-scale map through a window is a metaphor for the first navigation principle. The structure of an entire subsystem is typically much larger than the screen of a workstation. Therefore, at any one time only a small part of the current subsystem structure is visible. However, the window can be moved to another part of the document using the horizontal and vertical scroll bars and, hence, any one point of a document can be inspected.

Variable-scale map

A metaphor for the second navigation principle is browsing through a stack of related maps. It involves the selection of a subarea, "zooming in" (*i.e.*, changing the scale) on that area and, hence, revealing a more detailed map. The subsystems of a software system typically form a hierarchy (*i.e.*, a series of layers of abstract machines). Such hierarchies can easily be represented using nested documents; opening an icon has the effect of moving one level down in the hierarchy (zooming in) whereas closing a document means moving one level up in the hierarchy (zooming out).

Table of contents

Skimming the table of contents of a world atlas is a metaphor for the third navigation principle. The reader can easily determine how the world is structured.

He or she can then swiftly look up the map of a particular country via its page number. The table of contents corresponds to the top level of the graph editor which exhibits a macroscopic view of the system. The subsystem structure is a directed graph whose nodes are icons. The arcs represent the relationship between the subsystems.

The Rigi editor provides algorithms for the efficient compilation of module interfaces, Hood *et al.* (1986); Tilley and Müller (1988). The global interface analysis algorithms analyze, predict, and limit the effects of an editing change in a basic interface of a software system on the entire system. The algorithms improve on the deficiencies of the traditional compilation rule found in strongly-typed, separately-compiled programming languages, Swinehart *et al.* (1986), which forces the recompilation of many definition and implementation modules that may be unaffected by a change in a basic interface. The algorithms assume a software development environment like K2 which provides efficient access to the compilation dependencies and the module interfaces of the components being implemented. Thus, basic interfaces do not need to be frozen before they are sufficiently explored and tested.

The Rigi editor encourages exploratory programming; the designers can rapidly and easily explore different decompositions of a system and evaluate their merits by comparing their visual representations. Two prototypes of the Rigi editor, both written in the programming language *C*, have been implemented on the Apple Macintosh^{TM}, Müller (1986), and on the Sun-3 Workstation^{TM}, Müller and Klashinsky (1988). The design of the graphics interface was strongly and directly influenced by the user interfaces employed on the Macintosh, the Sun-3 Workstation, and the Cedar programming environment, Teitelman (1984). The current prototype of the underlying project data base uses the *ndbm* data base provided with the Unix^{TM} operating system, Unix (1986), to store the graphs; large text documents and bit maps are stored in the Unix file system. However, future implementations of the repository might use a data base such as *GRAS*, Brandes and Lewerentz (1986), that is specifically tailored towards the representation of graphs.

5. AUTODOC

5.1 Specification Documents

Autodoc is a structure-oriented editor for specification documents. It supports the *Software Cost Reduction* methodology (SCR), Parnas and Clements (1986). The SCR methodology is based on three types of specification documents: (1) system or requirements specifications, (2) interface, module, or black box specifications, and (3) internal or program specifications.

Requirements Specifications, Henninger (1980), describe the required behavior of a system in terms of its observable inputs and outputs. Each input and output is uniquely named and precisely described. A function is defined for each output, describing its value in terms of the inputs. Considerable attention is given to timing and accuracy constraints, and to error handling. To ease the maintenance task, expected changes in the system's behavior are recorded. For systems too large to be implemented by a single person, the development task is decomposed into work assignments called *modules.*

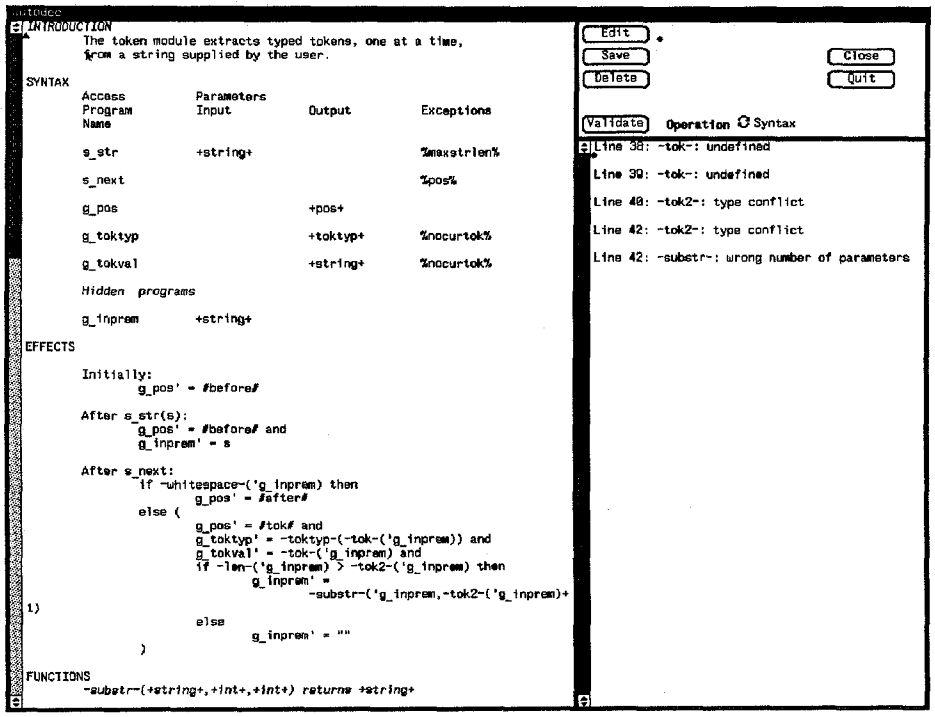
Module Specifications, Parnas (1972) and Hoffman (1985), describe, for each module, the services that the module provides, in terms of calls made on, and values returned by the module. Thus, a module is specified as a "black box" which hides its internal data structures and algorithms. Upon completion of a model specification the *module state* (*i.e.*, internal data structures) is defined.

Program Specifications, Mills (1975), indicate the relationship between the calls supported by the module and the module state. For each call, the new module state is expressed in terms of the module state prior to the call. In addition, the return values are expressed in terms of the module state.

The structure, syntax and semantics of the three document types are based on first-order logic, set theory, and finite state machines.

5.2 The User's view Of Autodoc

The Autodoc system exploits the interactive high-resolution graphics provided by modern workstations. The designers interact through a menu-oriented interface that displays icons, diagrams, and text in multiple windows on the screen. Autodoc provides support for specification (1) *reading* by providing simultaneous screen access to a particular specification object and to objects related to it by use or definition, and by providing multiple views for different readers, (2) *writing* by providing templates for the standard documents and document objects, and (3) *error checking* by providing syntax and type checking for the family of specification languages. The current prototypes of Autodoc support SCR requirements and interface specifications. Figure 2 below depicts an interface specification document as constructed by Autodoc.



```
INTRODUCTION
        The token module extracts typed tokens, one at a time,
        from a string supplied by the user.

SYNTAX
        Access          Parameters
        Program         Input           Output          Exceptions
        Name

        s_str           +string+                        %maxstrlen%

        s_next                                          %pos%

        g_pos                           +pos+

        g_toktyp                        +toktyp+        %nocurtok%

        g_tokval                        +string+        %nocurtok%

        Hidden  programs

        g_inprem        +string+

EFFECTS

        Initially:
                g_pos' = #before#

        After s_str(s):
                g_pos' = #before# and
                g_inprem' = s

        After s_next:
                if -whitespace-('g_inprem) then
                        g_pos' = #after#
                else (
                        g_pos' = #tok# and
                        g_toktyp' = -toktyp-(-tok-('g_inprem)) and
                        g_tokval' = -tok-('g_inprem) and
                        if -len-('g_inprem) > -tok2-('g_inprem) then
                                g_inprem' =
                                        -substr-('g_inprem,-tok2-('g_inprem)+
1)
                        else
                                g_inprem' = ""
                )

FUNCTIONS
        -substr-(+string+,+int+,+int+) returns +string+
```

```
Line 38: -tok-: undefined

Line 39: -tok-: undefined

Line 40: -tok2-: type conflict

Line 42: -tok2-: type conflict

Line 42: -substr-: wrong number of parameters
```

Figure 2. SCR Interface Specification

5.3 The Implementation Of Autodoc

This section illustrates how a software system implemented under a traditional software development environment can be converted to a K2 controlled software system.

The first prototypes of Autodoc were implemented using the Unix software development environment, Kernighan and Pike (1984). The software structure of these prototypes is recorded and maintained by means of the hierarchical file system, the symbolic file linking facility, and the make utility of the Unix programming environment. The *software structure* of Autodoc consists of a *module structure* and a *build structure*, Parnas (1974). The module structure is a tree in which the nodes and edges denote modules and module relationships, respectively. This data structure is implemented as a directory tree in the hierarchical file system. The root node represents the entire Autodoc system and stores its requirements specification. The interior nodes represent conceptual submodules and the leaf nodes contain actual module implementations. Each leaf node contains five files storing the module's interface specification, the source code of the module's implementation, the source code of a "test driver" designed to exercise the module's implementation, the source code of a "stub" which allows clients to compile and link successfully before the module is implemented, and a "makefile" to build the entire component. The module structure of Autodoc is depicted in Figure 3. It consists of five interior nodes and 17 leaf nodes, and stores 9,000 lines of source code. The build structure is stored in makefiles. It represents compilation and link dependencies among files. Multiple developers can work on the same software structure by copying the files to be modified and by establishing symbolic links to the remaining files.

This model of software structure was effective for the implementation of Autodoc and several other projects. However, the implementation under Unix exhibited three deficiencies.

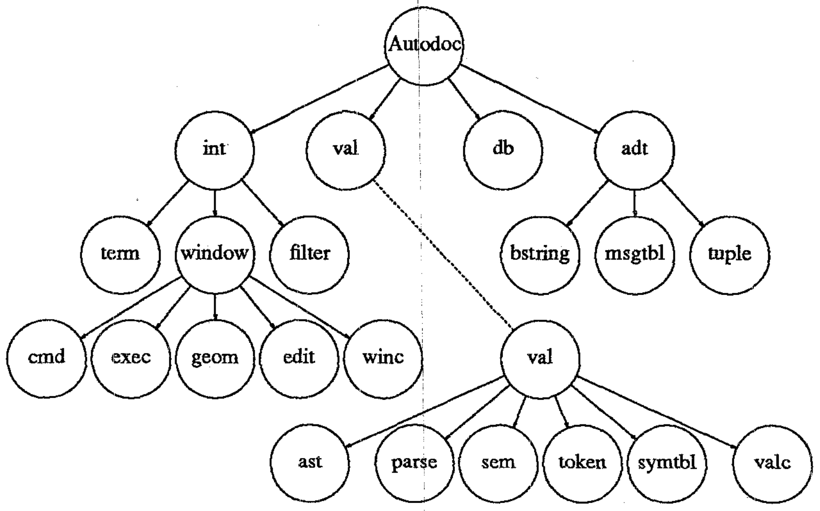


Figure 3. Module structure of Autodoc

(1) An overview of the module structure is only obtained by traversing the directory tree.
(2) The maintenance of the makefiles is tedious and error-prone particularly if the source code is distributed over many directories.
(3) Even for a small development team, many files are duplicated and a large number of symbolic links are required to implement the module structure.

The graph model of K2 is designed to represent software structures as used in the current Autodoc implementation. The nodes and edges of the module and build structure are modeled by object and dependency classes, respectively. The directory tree is directly represented by the aggregation hierarchy formed by the object classes. The files stored at the leaf nodes are system description documents. The module and build structures are browsed and edited using the graph editor. An overview of these graphs can easily be obtained using the three navigation principles of the graph editor. Forked development is supported through the linking and editing operations of the graph editor, which affect entire aggregation hierarchies, and through its version control facilities.

6. Mk*

6.1 Description of the Mk tools*

The *Compiler Development Environment* comprises a series of editors and generators to assist in developing translators. This collection of compiler tools is called *MkStar*, or Mk* for short. K2 performs numerous translations between system description documents; in particular, parsing and unparsing between display data structures (text and graphics) and internal data structures (trees and graphs). In addition, the envisaged document compilers that operate on the project data base will be generated using the Mk* tool series. Thus, Mk* is an important ingredient of the K2 implementation. The tool series is envisaged as initially having five main components. These are MkScan, MkParse, MkTypeCheck, MkSemantic, and MkCodeGen, as depicted in Figure 4 below. Each tool consists of an editor/generator pair responsible for maintaining one compiler module (or phase). The editor function of the tool allows the user to create or change the requirements for the corresponding compiler phase. After the user has finished specifying a consistent set of requirements, the generator function of the tool can be invoked to translate the description into program source code. The generator function of each tool creates a separate module. Linking the five modules together results in a complete compiler.

MkScan

MkScan is an interactive tool for generating lexical analyzers, Horspool and Levy (1987). The current implementation uses a series of simple menus to enable the user to select token types and their lexical structures. A prototype version of MkScan has been in use for some time as a replacement for the standard *lex* tool provided with the Unix operating system.

MkParse

The goal of a parser is to read a sequence of tokens and create a representation of the parse tree. The particular representation adopted for use in the generated compiler is that of an abstract syntax tree. The editor function of MkParse is to accept a specification of a mapping between the concrete syntax (which is conventionally represented by production rules in BNF notation) and the abstract syntax. The generator function of MkParse produces a combined parser and tree builder. A prototype version of MkParse that lacks a screen-oriented interface but which permits interactive, incremental updates to the grammar is already available. The chief benefit of this interactive parser generator is that the user is informed immediately if an unsuitable production rule is added to the grammar. The instant feedback speeds up the processor of grammar debugging.

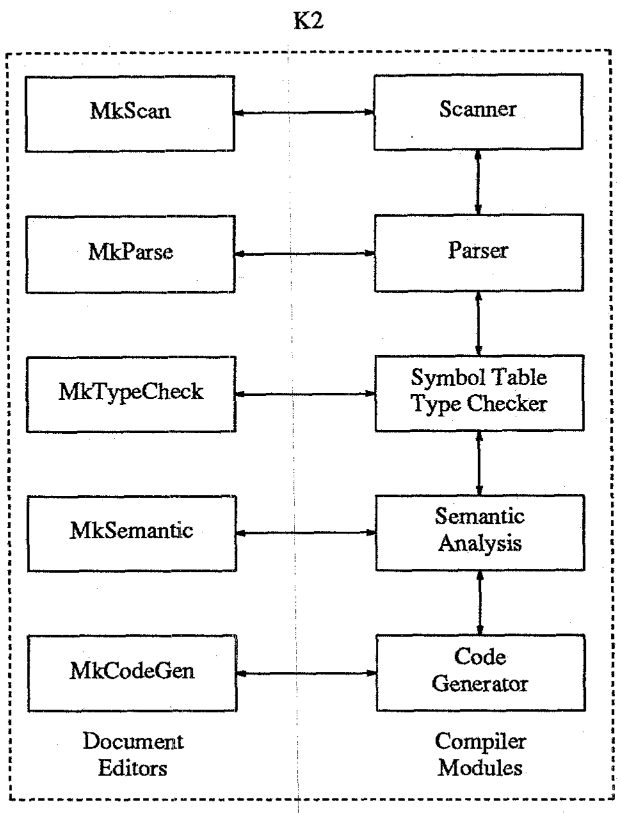


Figure 4. Mk* tool series.

MkTypeCheck

The generator part of MkTypeCheck will create a program that traverses an abstract syntax tree and performs type checking. Since type checking requires knowledge of the data types of identifiers used in a subject program, the tool must also handle associations between uses of identifiers and the definitions of those identifiers. This is the traditional task of the symbol table routines in a compiler. It is intended that the user of MkTypeCheck should view a node in the abstract syntax tree as an operator which is to be applied to its operands, namely to its children in the tree structure. The user is required to supply type signatures for each operator. If the operator is overloaded, there may be several alternate signatures. For example, if the programming language is Pascal, the user might specify the following type signatures for the '+' operator tree node as follows.

$$\begin{array}{ll}
\text{int} \times \text{int} & \longrightarrow \text{int} \\
\text{int} \times \text{real} & \longrightarrow \text{real} \\
\text{real} \times \text{int} & \longrightarrow \text{real} \\
\text{real} \times \text{real} & \longrightarrow \text{real} \\
\text{set(T)} \times \text{set(T)} & \longrightarrow \text{set(T)}
\end{array}$$

In this example lower-case names denote types (or type constructors) built into the language and upper-case names represent unknown types. Symbol table operations and checking of identifier classes are specified by means of extensions to the type signature notation.

MkSemantic

Semantic checking in addition to type checking is usually required. For example, a Pascal compiler should check that labels in a case statement are not duplicated. MkSemantic will generate a program to traverse the abstract syntax tree and perform whatever additional checking is specified by the user. In addition, attribute values that may be needed to simplify the task of code generation may also be computed and attached to nodes in the tree. Since the nature of the additional semantic checking and attribute computations is very language specific, the tool must necessarily be quite general. An attribute grammar scheme will be used for this purpose, tailored towards abstract syntax rather than concrete syntax rules.

MkCodeGen

The generator function of the tool will create a program to traverse the abstract syntax tree and generate object code for a desired target computer. The methodology will be based on the machine-independent strategy described in, Horspool and Scheunemann, (1985); Horspool, (1987). The editor function of MkCodeGen has to handle three kinds of information. The first is a set of storage resources that will be available on the target computer. The second is a collection of conversion templates that implement transfers from one storage resource to another in the target computer. The third is a set of code templates that implement the effect of each operation in the abstract syntax tree on the target computer. The two kinds of templates incorporate attribute evaluation rules and therefore permit generality when it is required.

If it is appropriate for the code generator to produce output in assembly language format, the tool should easily be capable of defining a complete code generator. Output in a format like that of relocatable binary code would either require additional routines to be supplied by the user or require an extra tool and another document type to define the object file format of the target computer.

By using a simple, consistent user-interface and by providing interactive feedback to erroneous or conflicting specifications, the effort needed to create a new compiler for a small language is significantly reduced. Larger languages with awkward semantics will, of course, require more effort but should still be much easier to develop than with existing compiler development tools like *lex* and *yacc*, Aho *et al.* (1986).

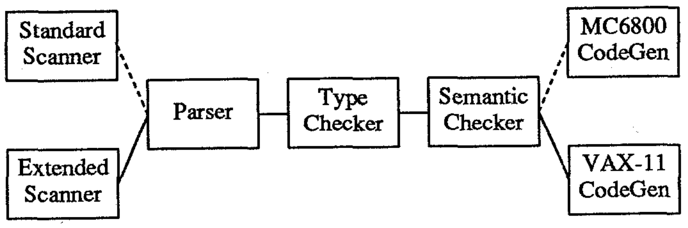


Figure 5. Mk* compiler variants

The five Mk* tools listed above are viewed as forming the minimum set of tools that would be useful. Extra tools to enable the user to specify tree transformations, machine-independent code optimization, and peephole optimization would certainly be desirable. And a separate tool responsible for specifying and generating the compiler's symbol table module would provide more flexibility than the current approach.

*6.2 The User's View Of Mk**

The translator project would be presented to the K2 user as an icon on the screen. Selecting this icon will cause the icon to be replaced on the screen by a group of icons one for each version of the compiler that has been previously created by the user. One such version represents a compiler variant. Selecting the icon for a translator variant causes a group of icons to be displayed. There are icons for all revisions of the compiler modules belonging to the project. The particular versions of the modules that form a complete compiler variant can easily be identified as illustrated in Figure 5.

Each of the icons (project, variant, module, or revision) represents a K2 *system description document*. If the user selects a scanner module icon, MkScan will be invoked as an editor on the scanner description document. The scanner description document contains all the lexical token definitions. If the generator function of MkScan is invoked, either explicitly by the user or automatically when a variant of the compiler is built, executable code for the scanner will be added to the document. The other Mk* tools manage their corresponding documents in a similar manner.

K2 version control facilities are used to enforce interface requirements between the translator components and to represent multiple versions of the module (e.g., the code generator specification might be tailor to different target computers). It is easy to imagine that multiple versions of other modules would exist while a compiler is under development. And different versions of the modules will almost certainly have different interfacing requirements. For example, one version of a MkScan document might define the operator symbol '&' while another version might define the keyword 'AND', yet both '&' and 'AND' might be intended to correspond to the grammar symbol 'and' in the MkParse document. Thus, additional tools to match interfaces are required. Both MkScan and MkParse assume that symbols are represented by numeric codes. But neither tool actually assigns the numbers. That is the responsibility of the interface checker, which will, by default, match symbols with the same textual representation. But it will ask the user for help whenever the correspondences are not obvious and will ask the user to confirm the default choices. The interface checker tools are automatically invoked whenever the user creates or modifies a compiler variant.

7. SUMMARY

The K2 software development environment provides effective mechanisms for expressing, presenting, and manipulating all information accumulated during the construction and maintenance of large, integrated, evolving software systems. K2 uses a graph model, object and dependency classes, and abstraction mechanisms to model the software building blocks and their dependencies. The model is a blueprint for the project data base which stores all the development information in system description documents. The data base is browsed and edited using the Rigi editor. This tool comprises an integrated collection of graph and structure-oriented editors to maintain the different types of system description documents. K2 will also include a set of document compilers, which browse the project data base, to produce typeset documentation at various levels of detail. The Mk* tool series allows the designers of K2 to experiment with various flavours of the document compilers and intermediate representations of the system description documents.

System description documents are intended to support software projects from the early specification phases through the long-term maintenance phases which sets this research apart from previous work on software development environments. In particular, Autodoc provides automated support for the construction of specification documents. The designer can also rapidly explore alternative system structures and, is therefore able to simplify and improve the system structure. Simplification, in turn, increases the reliability of the system. The integrators can check the system for consistency and completeness and, hence, are able to identify ambiguities. The maintainers can quickly analyze differences between versions and are thus able to predict how a change to one component affects the entire system.

ACKNOWLEDGEMENTS

This paper is a revision and extension of an earlier version, "K2 — A Software Development Environment for Programming-in-the-large," which appeared in the Proceedings of the Conference on Intelligence Integration, CIPS Edmonton '87 (Edmonton, Alberta, November 16-19), pp. 62-68, November 1987. K2 or Godwin Austen is a mountain in the Karakoram range in the Himalayas and Rigi is a mountain in the central part of Switzerland.

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada. The authors also would like to thank our students Alan Doheny, Michael Hayes, Alan Idler, Karl Klashinsky, Jingsong Liu, Yimin Shen, Scott Tilley, Yabo Wang, Jianguo Zhang, and Qing Zhao who all contributed to the implementation of the various prototypes.

REFERENCES

1. Aho, A.V., R. Sethi and J.D. Ullman, 1986, *Compilers — Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass.
2. Bentley, J.L. and D.E. Knuth, (1986), "Programming Pearls — Literate Programming," *Comm. ACM*, 29(5), pp. 364-369.
3. Bentley, J.L., D.E. Knuth and D. McIlroy, (1986), "Programming Pearls — A Literate Program," *Comm. ACM*, 29(6), pp. 471-483.
4. Brandes, T. and K. Lewerentz, (1985), "GRAS: A Non-standard Data Base System within a Software Development Environment." In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-large*, (Harwichport, Mass., June 9-12), pp. 113-121.
5. Brodie, M.L., J. Mylopoulos and J.W. Schmidt (eds.), (1984), *On Conceptual Modelling Topic in Information Systems*, Springer-Verlag.
6. Clark, D.D., (1985), "The Structuring of Systems using Upcalls." In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, (Orcas Island, WA, December 1-4), pp. 171-180.
7. Conklin, J., (1987), "Hypertext: An Introduction and Survey," *IEEE Computer*, 20(9), pp. 17-41.
8. Cooper, K., K. Kennedy and L. Torczon, (1986), "The Impact of Interprocedural Analysis and Optimization in the $\Re^n$ Programming Environment," *ACM Transactions on Programming Languages and Systems*, 8(4), pp. 491-523.
9. DeRemer, F.L. and H.H. Kron, (1976), "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, SE-2(2), pp. 80-86.
10. Goldberg, A., (1987), "Programmer as Reader," *IEEE Software*, 4(5), pp. 62-70.
11. Habermann, A.N. and D. Notkin, (1986), "Gandalf: Software Development Environments," *IEEE Transactions and Software Engineering*, SE-12(12), pp. 1117-1127.
12. Henninger, K.L., (1980), "Specifying Software Requirements for Complex Systems: New Techniques and their Application," *IEEE Transactions on Software Engineering*, SE-6(1), pp. 2-13.
13. Hoffman, D.M., (1985), "The Trace Specification of Communication Protocols," *IEEE Transactions on Computers*, C-34(12), pp. 1102-1113.
14. Hood, R., K. Kennedy and H.A. Müller, (1987), "Efficient Recompilation of Module Interfaces in a Software Development Environment." In *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, (Palo Alto, CA; December 9-11). In *ACM SIGPLAN Notices*, 22(1), pp. 180-189.
15. Horspool, R.N., (1987), "An Alternative to the Graham-Glanville Code Generation Method," *IEEE Software*, 4(3), pp. 33-39.
16. Horspool, R.N. and M.R. Levy, (1987), "Mkscan — An Interactive Scanner Generator," *Software — Practice and Experience*, 17(6), pp. 369-378.

17. Horspool, R.N. and A. Scheunemann, (1985), "Automating the Selection of Code Templates," *Software — Practice and Experience*, 15(5), pp. 503-514.

18. Kernighan, B.W. and R. Pike, (1984), *The Unix Programming Environment*, Prentice-Hall.

19. Leblang, D.B., R.P. Chase and G.D. McLean, (1985), "The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts." In *Proceedings of the IEEE Conference on Workstations*, (San Jose, CA).

20. Litchman, Z.L., (1986), "Generation and Consistency Checking of Design and Program Structures," *IEEE Transactions on Software Engineering*, SE-12(1), pp. 172-181.

21. Mills, H.D., (1975), "The New Math of Computer Programming." *Comm. ACM*, 18(1), pp. 43-48.

22. Müller, H.A., (1986), "Rigi — A Model for Software System Construction, Integration, and Evolution based on Module Interface Specifications." Ph.D. Thesis, Rice University, Houston, Texas, COMP TR86-36.

23. Müller, H.A. and K. Klashinsky, (1988), "Rigi — A System for Programming-in-the-large." In *Proceedings of the 10th International Conference on Software Engineering (ICSE)*, (Raffles City, Singapore, April 11-15), pp. 80-86.

24. Notkin, D. *et al.* (1985), "Special Issue on the Gandalf Project," *Journal of Systems and Software*, 5(2).

25. Parnas, D.L., (1972), "A Technique for Software Module Specification with Examples," *Comm. ACM*, 15(5), pp. 330-336.

26. Parnas, D.L., (1978), "On a "Buzzword": Hierarchical Structure," *IFIP 1974*, pp. 336-339, 1974. And in *Programming Methodology A Collection of Articles by Members of IFIP WG2.3*, D. Gries, (ed). pp.335-342, Springer-Verlag.

27. Parnas, D.L., P.C. Clements and D.M. Weiss, (1984), "The Modular Structure of Complex Systems," *IEEE Transactions on Software Engineering*, SE-11(3), pp. 259-266, March 1985. And in *Proceedings 7th International Conference on Software Engineering*, (Orlando, FL, March 26-29), (IEEE Catalog No. 84CH2011-5), PP.408-419, 1984.

28. Parnas, D.L. and P.C. Clements, (1986), "A Rational Design Process: How and Why to Fake It," *IEEE Transactions on Software Engineering*, SE-12(2), pp. 251-257.

29. Reiss, S.P., (1984), "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering*, SE-11(3), pp. 276-284, March 1985. And in *Proceedings 7th International Conference on Software Engineering*, (Orlando, FL, March 26-29), (IEEE Catalog No. 84CH2011-5), pp. 324-333, 1984.

30. Swinehart D.C., P.T. Zellweger, R.B. Hagmann and R.J. Beach, (1986), "A Structural View of the Cedar Programming Environment," *ACM Transactions on Programming Languages and Systems*, 8(4), pp. 419-490.

31. Teitelman, W., (1984), "A Tour Through Cedar," *IEEE Software*, 1(2), pp. 44-73, April, 1984. And in *IEEE Transactions on Software Engineering*, SE-11(3), pp. 285-301, March 1985. And in *Proceedings 7th International Conference on Software Engineering*, (Orlando, FL), (IEEE Catalog No. 84CH2011-5), pp. 181-197, 1984.

32. Tichy, W.F., (1979), "Software Control Based on Module Interconnection." In *Proceedings of the 4th International Conference on Software Engineering*, pp. 29-41, (IEEE Catalog No. 79CH2011-5).

33. Tichy, W.F., (1986), "Smart Recompilation," *ACM Transactions on Programming Languages and Systems*, 8(3), pp. 273-291.

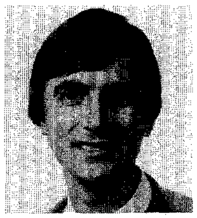
34. Tilley, S. and H.A. Müller, (1988), "Changing Module Interfaces in a Software Development Environment." In *Proceedings of the Sixth National Conference on Ada Technology*, (Arlington, Virginia, March 14-17), pp. 500-507.

35. Unix (1986) 4.3bsd system. On-line manual entry for "ndbm".

36. Winograd, T., (1979), "Beyond Programming Languages," *Comm. ACM*, 22(7), pp. 391-401, July 1979. And in *Interactive Programming Environments*, D.R. Barstow, H.E. Shrobe and E. Sandewall, (eds.), pp. 517-534, McGraw-Hill, 1984.

Hausi A. Müller received a Diploma Degree in Electrical Engineering from the Swiss Federal Institute of Technology (ETH), Zürich in 1979 and the M.S. and Ph.D. degrees in computer science from Rice University, Houston, Texas in 1984 and 1986 respectively. From 1979 to 1982 he worked as a software engineer for Brown Boveri in Baden, Switzerland. He is currently Assistant Professor of Computer Science at the University of Victoria. His research interests include software engineering, programming-in-the-large, programming languages, graph algorithms and computer graphics.

Daniel Hoffman received the B.A. degree in mathematics from the State University of New York, Binghampton in 1974 and the M.S. and Ph.D. degrees in computer science in 1981 and 1984 respectively from the University of North Carolina, Chapel Hill. From 1974 to 1979 he worked as a commercial Programmer/Analyst. He is currently Assistant Professor of Computer Science at the University of Victoria. His research area is software engineering, with emphasis on software specification.



Nigel Horspool received the B.A. degree from the University of Cambridge in 1969, and the M.S. and Ph.D. degrees from the University of Toronto in 1972 and 1976. He was both an Assistant and an Associate Professor at McGill University before moving to the University of Victoria in 1983, where he is currently a Professor. He is interested in many aspects of software systems, especially compilers, and has authored a book on the Berkeley Unix operating system.

Michael Levy received his Batchelor's and Master's degrees from the University of the Witwatersrand, and his Ph.D. degree from the University of Waterloo in Ontario. His is currently an Associate Professor of Computer Science at the University of Victoria. His main research interests are programming languages and type theory.