This is an extended version of the paper published in the *Proc. of the International Symposium on Applied Corporate Computing* - ISACC94, pp. 147-156, Monterrey, **Mexico**, Oct. 1994.

Software Development Environment for Concurrent Design and Maintenance of Complex Research Projects

Boris Stilman
Department of Computer Science & Engineering, University of Colorado at Denver
Campus Box 109, Denver, CO 80217-3364
E-mail: bstilman@carbon.cudenver.edu

Abstract—A Software Development Environment called PROGRAMMERS WORKBENCH (PW) is considered in this paper. It is intended to support concurrent software development and maintenance of large-scale research projects. Such projects usually involve an extended prototype period for investigation and improvement of the algorithm on which they are based. This period often includes even the maintenance phase, i.e., the entire software life cycle. Thus, each prototype must be extremely flexible to provide multiple redesigns. The environment supports concurrent development and multiple redevelopment iterations. It provides support of individual software development skills by combining independence of a software designer acting alone and strict discipline of cooperative research projects. The PW was implemented on IBM hardware. It has been used in several research institutions for the development and maintenance of large-scale artificial intelligence projects.

1. INTRODUCTION.

Many software design projects require extreme flexibility. First, we are concerned with research projects. Although requirements specifications might be clear and well defined, the design specifications and the algorithm are vague, because both of them usually are subjects for research. Of course, we can not develop software following a vague algorithm. Thus we have to define the first version of the algorithm, design the first software prototype, investigate this prototype and find a way for improvement of the initial algorithm or even to change it entirely. This brings us to the second iteration of the design process: we have to design the second version of the algorithm, prototype, etc. This model is analogous to the spiral software engineering model [1] with the following differences. First, we should not redefine requirements specifications after each iteration: they are stable. Second, we should not perform risk analysis, because our purpose is different. Instead, we have to conduct prototype investigation to test the ideas on which the algorithm is based. We have to improve these ideas and implement them in the new version. Obviously, it could be better to investigate the algorithm theoretically but for complex projects it is usually impossible.

Following this software design model we can start each new iteration almost from scratch, i.e., from requirements specifications, and redesign everything. Unfortunately, it may extend the design process dramatically and eventually cause the failure of the entire project. A different approach is to redesign each new prototype from the previous one. Thus, we have to make every prototype (and its components) extremely flexible and in some sense reusable [2]. We need a problem-oriented building set of reusable building blocks of different sizes. The harder the skeleton of the old building, the easier to perform future redesign. Sometimes, very rarely, even the skeleton should be rebuilt. To support flexible, changeable prototypes we have to provide a hard structure for the applications through structured design and implementation, and what is more important, to keep this hard structure during the entire software life cycle. This problem is closely related to the notorious software maintainability problem that is considered as desperate problem for all types of software projects (not only for the research ones) [3].

The next problem, which complicates our discussion substantially, is the problem of concurrent development. A series of flexible prototypes should be designed in a close cooperation of software research engineers. The question is how to support the design of a large structured prototype by a software team. This might be accomplished by breaking down the design into sub-designs, by the support of the interaction and control of sub-designers, and, of course, by providing a strict design discipline. At the same time, team members often are bright researchers, or even simply bright software engineers whose creativity skills would naturally resist this all-embracing cooperation, control and discipline. How to reveal these natural skills, how to give individuals the freedom required for revealing their creativity? We must find a method to direct all their energies to the benefit of the project instead of the natural resistance to cooperation and discipline.

The support of *software flexibility*, *maintainability*, and *concurrent development* is the key point of our requirements for the design of a software development environment. Subordinate to these key points are software *run-time efficiency, portability, software correctness* and *reliability*.

Complex research projects usually demand so much of the hardware that SDE overhead expenses should be reduced to a minimum. Moreover, the environment should support the development of applications with the highest performance parameters.

The environment itself should be *portable*, employing for example the approach of the UNIX designers. The biggest hardware independent piece should be written in a higher level popular language, while the rest of the system, the hardware dependent piece, should use an assembly language, and would be rewritten in case of porting the system to new hardware. Obviously, the environment should support the development of extremely portable software, because porting to new equipment is the routine procedure for long-range research projects.

The environment should support the development of "*correct*" and *reliable* software to give a researcher the opportunity for investigation of algorithms of all the intermediate prototypes, avoiding interference of bugs as much as possible.

There are many different software environments [4, 5]; they have some advantages and disadvantages, and research in this field is the hot point of software engineering. In order to meet precisely the requirements presented above we have designed a new software development environment (SDE) PROGRAMMERS WORKBENCH.

The requirements listed above require further refinement in the form of the software development model. In this paper we consider such a model (Sections 2-5); then we present implementation issues.

2.TOP-DOWN DESIGN: VERSION APPROACH

In order o expedite the development of a software project we have to break it into parallel processes as early as possible. This way we can involve team members in simultaneous work. According to principles considered in the introduction, we are going to support software development beginning with the design stage. Thus, we have to introduce a parallel design model as well as parallel models for all the rest of development steps: implementation, testing and debugging, and maintenance. Our main software development model represents software development as the design of the sequence of prototypes, so all these parallel models should be included into the cycle and, moreover, nested into each other. The key element of this series of models is the notion of a version.

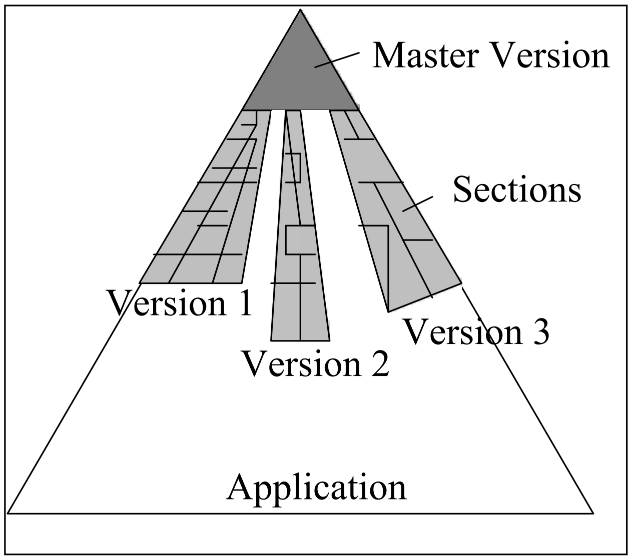


Fig. 1. *Top-down design: initial versions*

A *version* is an independent software unit containing design specifications (architectural and procedural), implementation information (source and object codes), testing/debugging information, specific debugging problem-oriented tools, and control information. A version is intended for the *personal development* by a software team member.

The process of the development of complex software project might generate tens or even hundreds of different versions. One of the main purposes of the environment is to provide the support for the design, cooperation, and control and design of these versions.

Here, we would like to emphasize that usually a typical version does not represent the entire project. At best, it is a rough draft only. Some of these versions, the designated versions, do represent it. This row of designated versions corresponds to the row of prototypes of our model.

Let us consider the creation of versions at the initial stage of software design (Fig. 1). We follow the top-down design model proposed in [6], a structured programming approach [7] and later proposed methods for the translation of data flows into design definitions [8]. The pyramid shown in Fig. 1 represents a design structure chart of the application to be developed. This is a hierarchical tree of boxes (modules) with the root (main module). Branches of this tree are the sequences of modules traced down from the top, such that each successor is subordinate to the predecessor. The set of modules (or sections) included into the top of this structure chart is called a *master version*. They represent subordinate sections of the highest level as well as, probably, some of the most important I/O sections of our application. This initial master version will be the common piece and, consequently, must be included into all the versions to be created.

Here is the point for the initial break of the design into sub-designs. We can separate some branches. Each branch should include its own sections as well as a common piece, the top sections. Such a branch is called an *initial version*. Versions shown in Fig. 1 have a common piece, the master version. However, it is not a restriction: they might have some other intersections as well, i.e., different common sections that are very important to be included into the group of versions, but not important enough to be included into the master version.

Obviously, we can break down our design not only at the top. Any node (box) of the structure chart tree can be considered as the break point. New versions to be created must include the master version, existing versions coming down into this node (or at least sections of one branch coming into this node) and one of the branches coming down from this node. The main reason for the creation of a new version is to initiate an independent software design process. When creating a new version we should take into account the semantics of this future version (as an independent set of sections) and the possible consequences of this (even temporary) separation.

3. VERSIONS IMPROVEMENT CYCLE

As we know the creation of a new version initiates an independent software development process. This process includes all the stages of software life cycle: design, implementation, testing/debugging, and maintenance. Even after the generation of an intermediate prototype some versions might continue their life cycle during the algorithm investigation. During their life cycle, versions can actively interact to each other. Next, we will consider different types of mutual interactions between versions and the outer environment.

While the creation of new versions happens rarely during software process, more often we have to include one of the existing versions into all other versions. Assume that at some stage of software development 9 different versions have been created (Fig. 2). Versions A1, A2, A3 were at the disposal of the designer A, versions B1, B2, B3 and C1, C2, C3 — at the disposal of the designers B and C, respectively. All the versions include the common piece, the master version. Subsequent development was conducted independently by each designer. It could happen that one of the versions, e.g., version A1, has "matured" enough to be designated a new master version. For example, version A1 has been fully designed, implemented and even tested and debugged. In this case it is feasible to make this designation and include this new master version into all the rest of the existing versions.

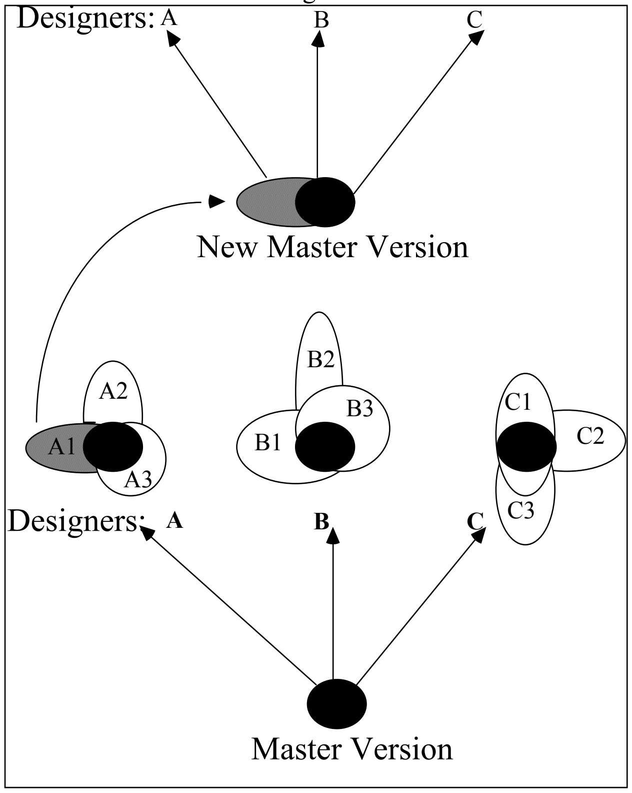


Fig. 2. *Versions development cycle*

The purpose of this inclusion is to bring our versions closer to the complete future prototype, to reduce missing pieces (which were substituted by stubs). Obviously, this version improvement should be done as soon as possible. This procedure is depicted in Fig. 2: version A1 is designated a new master version and included in all the versions of designers A, B, C. This procedure is called a *version improvement cycle*.

To avoid misunderstandings, we have to distinguish this version improvement cycle from the main development cycle resulting in the new prototype after each iteration. Each iteration of this main cycle includes multiple iterations of the version improvement cycle; the final version improvement creates the next prototype.

4. VERSIONS INTERACTION: MACROOPERATIONS

Along version improvement many other operations with versions are required to support cooperation and control of independent development processes. Here, we will consider *macrooperations* because they can be defined in terms of the whole versions without consideration of version inner structure. These macrooperations are intended to support close cooperation and control of software designers through automatic support of various interactions between versions (Fig. 3).

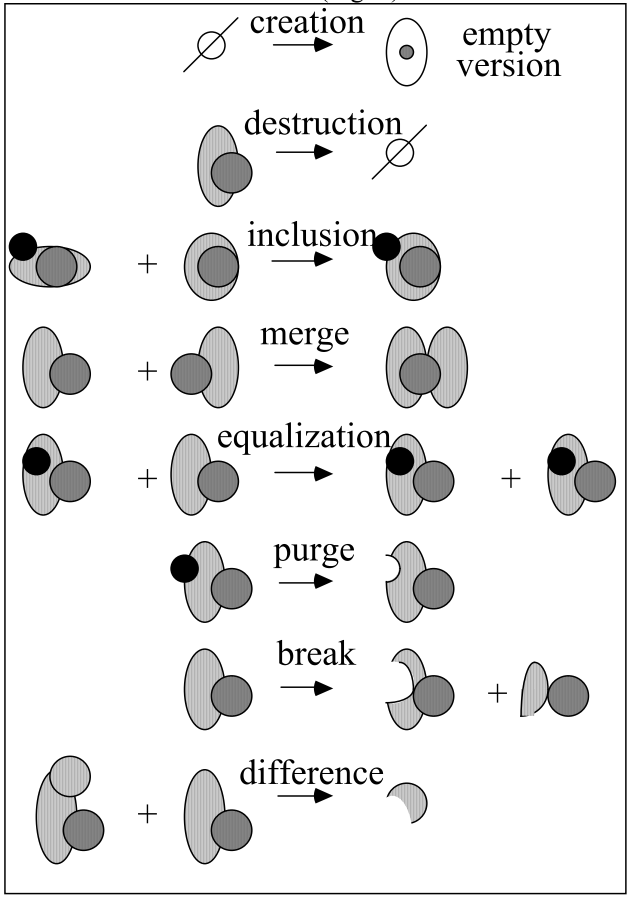


Fig. 3. *Versions interaction: macrooperations*

The first operation is called a *creation*. It takes place when we have to generate the *empty version*. This empty version does not include any application design sections but it has something, which is called a *kernel*. The kernel is a software unit that is included into every version. Its purpose is to support interaction of this version with the software development environment, to support all the operations, and actually to turn a set of design sections into the version. The kernel itself is invisible for a software designer, but after creation, the version, although empty, becomes "visible as independent object represented by name" in different lists generated by the environment.

The next operation, called a *destruction*, is very simple. The version considered as an object, which concluded its task, must be destroyed. The version that is subject for destruction is shown in Fig. 3 as a union of two figures, an ellipse and a circle. The circle represents the master version.

The operation called *inclusion* can be explained as follows. During the independent development of a version (see left argument in Fig. 3) some sections were designed, implemented and tested, while the entire version is not ready yet to substitute for the new master version. These sections are shown in Fig. 3 as black circles. In this case a manager can make a decision about inclusion of the finished piece (black circle) into other versions (right argument).

Sometimes independent development of some versions can not continue any longer. For example, comprehensive testing of one of the versions requires the development of complicated stubs to substitute for the sections of the other version. A different situation is as follows: independent development showed that separation of this version was unreasonable because the semantics of a separated piece was not well defined. In all such cases we have to merge this version with some other existing ones. Operation *merge* is shown in Fig. 3. Of course, the master version component will not be duplicated in the result of this operation. It is shown as one small circle in Fig. 3. The same operation is used for inclusion of the new master version into all other versions during the version improvement cycle.

After active development, a version might be left unchanged, actually preserved, for future reference, while its copy will stay under development. This way the preserved version will grow old: some sections will no longer be up to date. They will be changed in the active copy. That is why we need an operation of *equalization* of two versions. After this operation both versions will become identical by substituting aged sections for the updated copies (black circle in Fig. 3).

The operation of *purge* is intended to purge some sections (black circle) from a version. These sections proved their unsoundness in different versions and earlier were deleted from them. When it is decided to purge all the unsound sections from the rest of versions, it can be accomplished by the purge operation.

At some stage of the version development it may become apparent that this version should be broken into two different versions. It might happen when a designer decides that the current version is “overloaded” (got too big) and further development, e.g., testing, can not continue properly without a break. With the list of sections to be separated as an input, the operation *break* creates two new versions.

To keep track of the development of different versions and control cooperation of the designers, we have to compare versions in pairs to find a difference, a list of sections that are included in the first version and not included in the second. The operation *difference* performs this duty. The result of this operation is often used as an input for other operations, e.g., for the break.

The macrooperations considered above support interaction between versions, and in this way, cooperation of software developers. Below we consider *microoperations*, which support the development of each version in a version life cycle.

5. VERSION DEVELOPMENT: MICROOPERATIONS

The following operations are actually routine software development procedures but here they appear in the new framework of version development. They are called *microoperations*, because they require to interfere into the inner structure of a version or display some details of this structure. Microoperations are broken into three modes: design mode, execution mode and debug mode (Fig. 4).

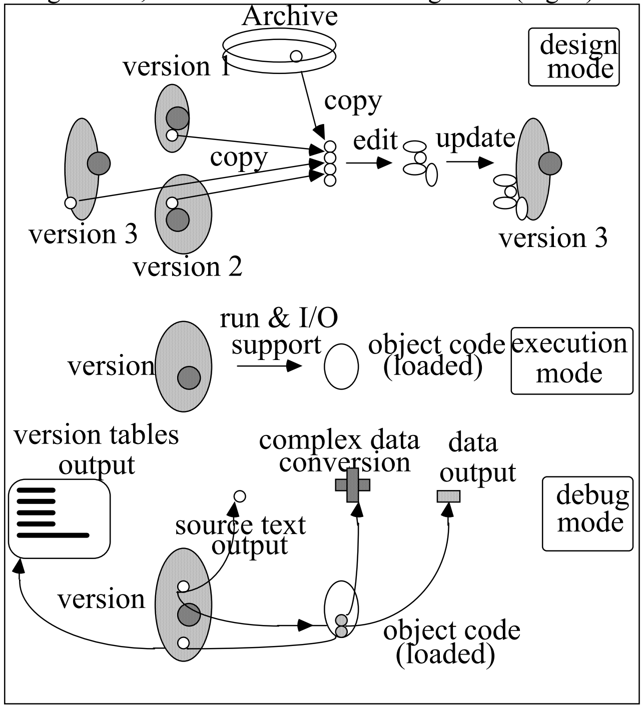


Fig. 4. *Version development: microoperations*

In the design mode a developer updates his version. He designs new sections of the structure chart, updates this chart, develops detailed design of sections, e.g., in pseudocode and in English, updates this design, develops sections source texts and updates them. First, he employs requirements specifications, draft pieces of design, pseudocode and source texts, which are kept in the SDE Archive (Fig. 4). Second, a developer can use existing versions as a valuable source of draft designs and source texts (see versions 1, 2, 3 in Fig.4). All this information can be copied into the developer’s work file for joint editing. Small white circles in Fig. 4 represent sections. After editing, the contents of sections and their number have been changed. Four sections shown in Fig. 4 are combined into one new section. Now we are prepared to update version 3. We should keep in mind that a version is a complex object and can not be updated instantly, as can be done with an editor’s work file on screen. Thus, version update requires a specific microoperation *update*. As a result of this, the prepared designs or source texts will be “compiled” into the version. For example, if source text of a section was edited and sent for a version update then the following changes will be accomplished. The SDE will update source text, object code, storage allocation and cross reference tables, debug options, and version control information.

Next, we are going to discuss an *execution mode* (Fig. 4). In this mode a version can be executed and tested as an executable unit. Every design section represented in the version by stub or source code is represented by object code as well. This code is generated by the SDE automatically. Thus, every section is executable, and the entire version can demonstrate the performance, which is close to the performance of the future prototype. (This closeness depends on the closeness of this version to the prototype.) The kernel of the version under SDE control should support multi-channel version I/O and allocation of physical terminals to the specified channels (I/O switches). It also provides run-time infinite loop protection, registers exceptional situations (e.g., errors), catches interrupts and, in this case, switches the version into debug mode.

The *debug mode* is intended to support detailed investigation of the version, looking for a source of exceptional situation registered at the execution mode. In this mode the execution is suspended or interrupted. In response to the developer's request SDE is able to display current trace of section calls, values of all the variables, data structures, current state of version files, all the source texts, cross reference tables (e.g., lists of subordinate and superordinate sections). In this mode a developer can also start a problem-oriented debugging subsystem to convert complex data to the higher level representation or generate standard "correct" data with automatic comparison with the registered values (Section 13).

In a short description of macro- and microoperations we outlined the basic principles of a software environment for cooperative design. In the following sections we will consider some details of implementation of these principles in the form of the SDE PROGRAMMERS WORKBENCH (PW). Early versions of this SDE are considered in [9, 10].

6. STRUCTURE OF PROGRAMMERS WORKBENCH

SDE consists of the following tools (Fig. 5):

— *PW Monitor*, an interactive subsystem which controls all other tools and communications with outer objects;

— *PW Editor,* a multi-user multi-window screen editor, which supports all the microoperations in design mode. It communicates with PW inner objects PW Librarian and PW Archive as well as with outer system software: Operating System (OS), OS Files Archive, and CICS Users Archive (Fig. 5);

— *PW Archive* of design and source text drafts, that are currently under development. It also contains source texts and object code of all the PW tools;

— *PW Librarian*, a subsystem for automatic support of all the macrooperations utilizing the version update microoperation. This subsystem manages the library of versions;

— *PW Run-Time Monitor*, a subsystem that supports microoperations in the execution mode;

— *PW Debug Monitor*, a subsystem that supports microoperations in the debug mode.

The PROGRAMMERS WORKBENCH is an integrated software development environment implemented with IBM hardware and software. Communications between SDE subsystems and with outer objects are shown in Fig. 5.

PW communicates with OS interactively emulating an operator console. OS files generated by versions during the execution (or other files) are accessible through PW Editor for screening in the design mode. Their update is permitted in the execution mode only. Communication with the standard IBM CICS Users Archive is fully interactive: CICS modules can be copied into the Editor's work file, updated and then reloaded into the CICS Archive.

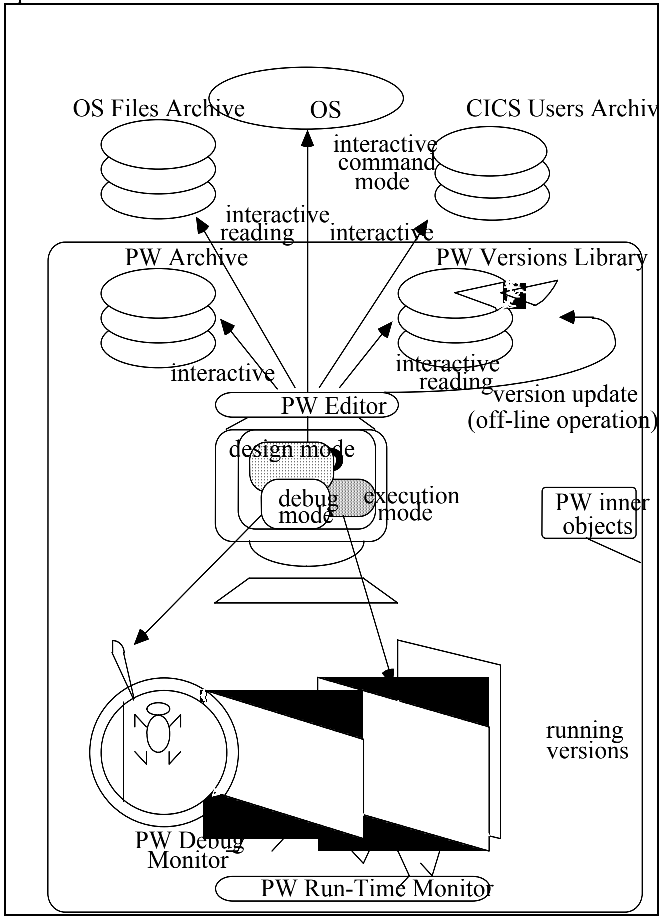


Fig. 5. *Structure and communications of PROGRAMMERS WORKBENCH*

Basically, inner PW communications support macro- and microoperations with versions. Macrooperations are complex and require a lot of processing. So, they are implemented in the off-line mode, while microoperations (except one) are fully interactive. The only exception is version update microoperation after source text editing. It is performed by PW Librarian in the off-line mode.

7. WORK PROCEDURES IN PW ENVIRONMENT

These procedures are depicted in Fig. 6. The design and implementation stages of the software development process are represented by the upper loop. Working with PW Editor a developer performs initial and detailed design of his version. A developer saves drafts of his design temporarily in PW Archive. The beginning of a version's own life cycle is as follows. After preparing the initial version as a set of pseudocode sections, stubs and section descriptions in English, a designer creates a version as a unit to be managed by SDE. This version update job being started by Editor's command, collects all the necessary source pieces from Editor's work file, Archive, from existing versions, combines all components, and creates new version or updates an existing one in the off-line mode. Now the designs and texts are present in the updated version; they form a structure that is available for the PW Editor for screening and further update. (The drafts saved temporarily in the Archive now must be deleted.) The subsequent design consists of repetitions of this loop.

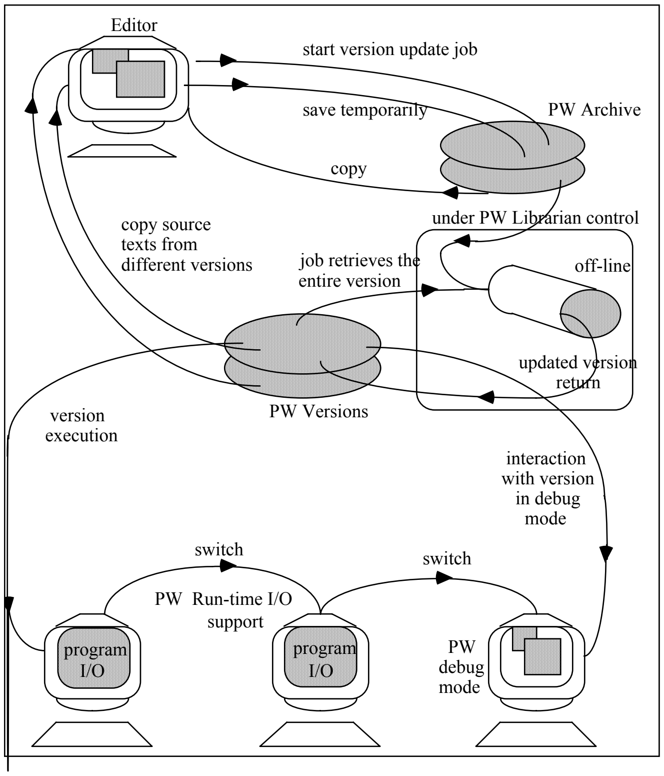


Fig. 6. *Work procedures in PW environment*

The implementation stage employs the same procedures. The only difference, practically invisible to the designer, is the execution of version update operation. SDE generates (or updates) object code, storage allocation and cross-reference tables, and checks syntactic structure of the entire version.

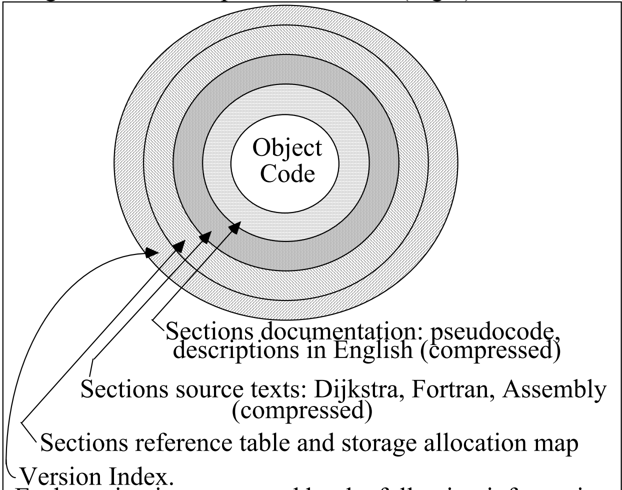
The subsequent testing and debugging steps are presented at the bottom in Fig. 6. In the execution mode running versions under control of the Run-Time Monitor can allocate different terminals for I/O, switch from one terminal to the other (from one channel to the other), or even to a dummy terminal, i.e., continue execution without output. After registration of an exceptional situation, SDE passes control to the Debug Monitor. It will support a debug session on the same terminal or can switch to a different one. All the information about the interrupted (or suspended) version is available for the designer in this mode: source texts, cross-reference storage allocation tables, and values of data structures. We will consider some details of these modes in Sections 12 and 13. Testing and debugging result in a version update, i.e., we return to the upper loop procedures (Fig. 6).

The SDE PW supports concurrent design providing full support of versions macrooperations. Their implementation is based on the *version update* microoperation (Fig.4). The PW Librarian prepares source texts (designs, descriptions in English) of the sections to be updated. A list of these sections is usually the input information for the Librarian. Sometimes the Librarian itself can generate this information, e.g., comparing two versions and computing the difference, the list of different sections (Fig. 3); then it can use this difference for the version update. The update itself is performed by deleting sections to be destroyed, by generating object code for the new sections or for the sections to be substituted, by linking all the sections and recomputing the tables.

The approach to software development supported by SDE PW does not require a completion of a current development step before the beginning of the next one. For example, in order to begin testing we do not have to finish implementation of the entire version. An unimplemented piece will be represented by stubs and pseudocode. It means that we can have a "version prototype" at the initial steps of the design. The following development of this prototype will result in a cyclic repetition of steps: design – implementation – testing – debugging – design. In software development practice in general we always have this loop, but usually it is poorly supported. This loop is essential at the maintenance stage of the software life cycle. Thus, SDE supports maintenance long before the completion of the development: the *entire software life cycle* is considered as a *maintenance process*.

8. VERSION STRUCTURE

A version is implemented as a partitioned data set regarding to IBM OS MVS(SVS). Each version consists of the partitions of the following five types: object code (load module), sections documentation (pseudocode, descriptions in English), sections source texts on Dijkstra language [7, 9, 13], Fortran, Assembly, sections reference table and storage allocation map, version index (Fig.7).


Object Code
Sections documentation: pseudocode, descriptions in English (compressed)
Sections source texts: Dijkstra, Fortran, Assembly (compressed)
Sections reference table and storage allocation map
Version Index.

Each section is represented by the following information

Section name	Creation date	Last source update date	Compiler type	Debug statements (hidden)	Last documentation update date	Debug section (flag)

Fig. 7. *Version structure* (*simplified*)

Every subroutine of the application corresponds to one section of the version, i.e., to one partition of documentation and to one partition of source texts. Here, *subroutine* means an arbitrary piece of text with the following first and last lines: the first line contains the word SUBROUTINE and the name coming next, the last line contains the word END. Thus, all the subroutines in Fortran (or Dijkstra) are separate sections as well as pieces of the text singled out intentionally, e.g., functions in Fortran, Assembly subprograms, and descriptions in English. Every section has its own unique name, a name of the subroutine. Sections with source texts and documentation are compressed.

The structure of the Version Index is shown in Fig. 7. Let us consider some details. *Creation date* means the date when this section was included into this version. *Compiler type* contains a symbol corresponding to the compiler which was used for generating object code of this section. It might be an optimizing or debugging Fortran compiler, or Assembler. Next Index item indicates the presence of instructions generated by source statements DEBUG in the section object code. Last Index item contains an indicator that this section is a debugging one (see Section 11).

This version structure allows implementation of all the macro- and microoperations considered above. For example, if we have to equalize two versions, the Librarian can find in the leading version all the sections updated later than the specified date. After that, it will collect their source texts (documentation) and update the aged version. Another example: in order to report about aged documentation, the Librarian can search Index and find all the sections where *last source update date* is greater than *last documentation update date*. Employing PW Editor we can copy Index, any source texts or documentation from the version into work files. It allows us to edit simultaneously two or more texts copied from different versions in different windows, then combine these texts and send a command to start version update job.

Next we consider an implementation of the version update operation, which is the major component of all the macrooperations.

9. VERSION UPDATE OPERATION

This operation can be initiated by the designer typing the PW Editor command or automatically by the Librarian itself as part of a macrooperation. For simplicity we will consider this operation without documentation update and without Assembly sections.

A typical version update operation has three input lists (Fig. 8). The first one consists of the names of sections to be updated in the version (or new sections). The second list (possibly empty) consists of the names of sections for DEBUG statements insertion. Each insertion is accompanied by the indication of specific type of these statements and activating conditions. The third list shown in Fig.8 between first and second lists is the list of source texts in Dijkstra or Fortran. All these lists were previously updated by the PW Editor or passed by Librarian from the previous steps of a macrooperation.

The next step procedure, CHOOSER, selects from the third list source texts according to the names from the first and second lists. CHOOSER inserts run-time parameters check statements into all the sections selected. Conditional and unconditional DEBUG statements are inserted into the sections from the second list. No statements inserted above will stay in the version source texts. However, the version object code will be updated and the version Index will have a pointer, which indicates the presence of instructions generated by DEBUG source statements in the section object code.

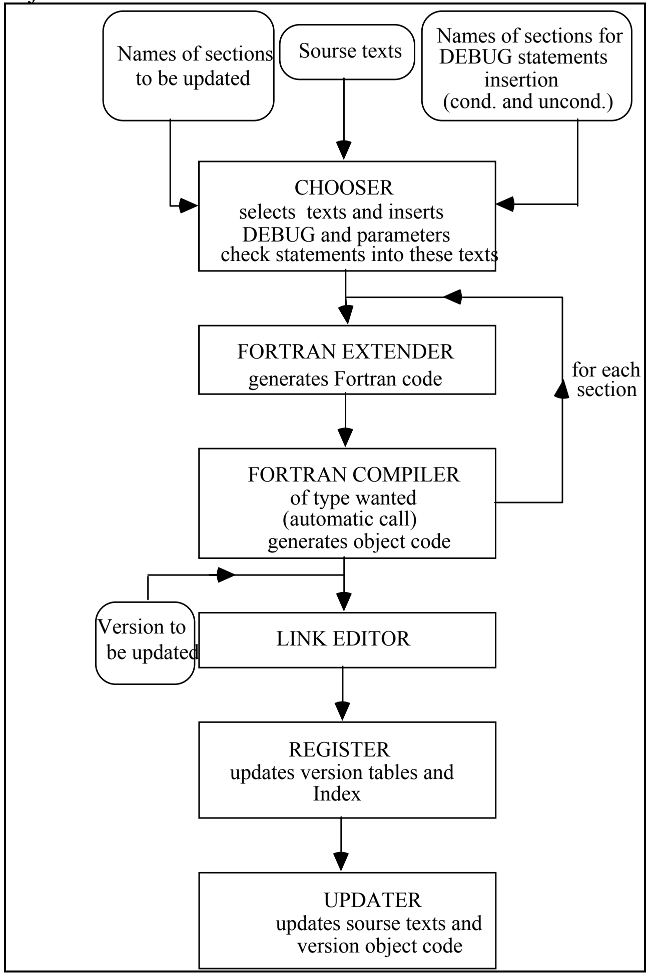

Names of sections to be updated
Sourse texts
Names of sections for DEBUG statements insertion (cond. and uncond.)
CHOOSER selects texts and inserts DEBUG and parameters check statements into these texts
FORTRAN EXTENDER generates Fortran code
for each section
FORTRAN COMPILER of type wanted (automatic call) generates object code
Version to be updated
LINK EDITOR
REGISTER updates version tables and Index
UPDATER updates sourse texts and version object code

Fig. 8. *Version update operation*

Source texts prepared in this way are passed to the FORTRAN EXTENDER. Basically, this procedure is the implementation of the preprocessor from the Dijkstra language to Fortran. It generates pure Fortran code and passes control to the IBM FORTRAN COMPILER. The type of compiler to be called (debugging or optimizing) is predetermined by the presence (or absence) of DEBUG statements in the current section. In case of the absence of DEBUG statements, only the optimizing compiler (with the highest level optimization mode) is used. Storage allocation tables are passed to the subsequent steps.

Object code sections generated by the compiler come into the IBM LINK EDITOR. This editor retrieves the version, i.e., partitioned file, to be updated. Then it inputs version object code (load module), and updates it. New load module and cross-reference tables generated are passed to the next procedure.

Having received storage allocation and cross-reference tables, the procedure REGISTER updates version tables and Index.

Next, procedure UPDATER updates section's source texts and object code. UPDATER compresses source texts received from the early steps and generates new sections (new partitions of the partitioned file). As usual for IBM OS partitioned data sets, new partitions are added to the end of the data set, while old partitions with the same names are destroyed. Before adding new partitions to the version, the Librarian checks the room in the data set for this operation and compresses this partitioned data set if necessary.

10. DOCUMENT UPDATE SERVICE

In order to meet the requirements of flexibility, reusability of some sections, maintainability and coordination of concurrent design stated in the introduction to this paper we have to keep a strict correspondence between all the version components. From Section 9 we can see that correspondence between source texts, object code, version tables and version Index is supported automatically. A manual source text update (after editing) can be saved in a version only through a general version update. Thus, by updating all the version components the Librarian keeps mutual correspondence between them. The problem is substantially harder for documentation update.

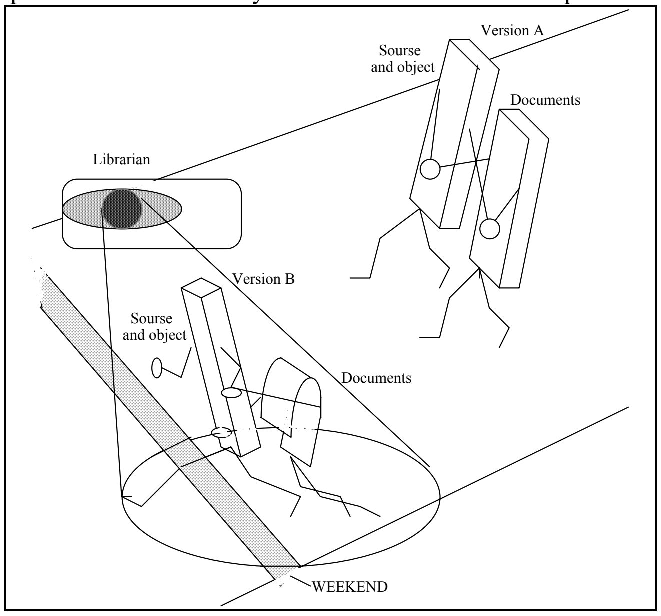


Fig. 9. *PW document update service*

We allow informal (but readable) documentation, e.g., pseudocode, descriptions in English; so its analysis and update are hard to accomplish automatically. To do that we would require an expert system with natural language analyzer, which was beyond the scope of this project. Thus, at this point we have to rely on a software designer: he must update documentation himself. While software development steps run successively, i.e., requirements analysis, design, implementation, etc., the documentation designed at each step is up to date. But after the first change at the implementation, testing or maintenance step and, especially, during algorithm investigation, we face a problem. It is hard to force the designer to update documentation before the source text update, especially if this change is only experimental and probably in a few minutes will be replaced by a new change or even by the return to the previous text. So we allow a temporary disparity of documentation and other version components, say, during a week. It should be a period while a designer can easily recreate the ideas which motivated him to make the change and reflect these ideas in documentation. However, no grace period is allowed after the deadline: a disordered version must be updated or suspended, i.e., SDE must make this version inaccessible for the following design (until documentation is updated).

The PW documentation update service is shown in Fig. 9. At the end of each week the Librarian scans all the versions checking sections with aged documentation. It scans version Index and compares last source and documentation update dates (Fig. 7). Disordered sections are printed out and handed to the designers. Then Librarian gives designers a grace period to accomplish update. After the deadline unupdated versions will be suspended.

11. FORTRAN EXTENDER

E. Dijkstra [11] developed an approach to software development based on the simultaneous program design and correctness proof. One of the most important features of this approach is the use of the language constructions whose semantics can be described formally in terms of predicate converters. The SDE Librarian includes FORTRAN EXTENDER (Fig. 8) based on Dijkstra control constructions [11]. This way we achieved a combination of understandable and readable source texts structure with the efficiency typical for Fortran compilers.

The Extender input language is an extension of Fortran. It includes Dijkstra constructions IF and DO. The syntax of the input language is as follows (assuming Fortran terms are known):

```
<program>::= [<subroutine name>] {<description
   statement>} {<executable statement> | <format
   statement>} <END statement>
<executable statement>::= <executable Fortran
   statement> | <IF statement> | <DO statement> | SKIP |
   ABEND
<IF statement>::=  IF<list of protected statements>FI
<DO statement>::= DO <list of protected statements>OD
<list of protected statements>::= <protected statement>
   {¤ <protected statement>}
<protected statement>::= <protector> {<executable
   statement> <format statement>
<protector>::= <simple protector> | <simple protector>
   {CAND<simple protector>} |
   <simple protector> {COR<simple protector>}
<simple protector>::= <Fortran Boolean expression> —>
   | ELSE —>
```

Fortran statements are written according standard Fortran rules. Simple protector is written in one or more lines. The continuation of the line is indicated according to Fortran rules. Extender's key words IF, FI, DO, OD, ¤, CAND, COR, SKIP, ABEND are written in a separate line each.

Basically the semantics of control constructions corresponds to [11, 12] and can be briefly described as follows.

Execution of the protected statement consists of the execution of its executable statements. It is allowed if the protector of this protected statement is true. Execution of the list of protected statements consists either in execution of one of the protected statements with true protector (which one of them is not defined) and output of the signal “success”, or output of the signal “failure” if all the protectors are false.

Execution of IF statement takes place as follows: first, list of protected statements, enclosed with brackets IF FI, is executed, then if “success” is signaled a control is passed to the next statement after FI, if the signal is “failure” then program execution is interrupted. Execution of DO statement also consists of the execution of the list of protected statements, then in case of “failure” signal, control is passed to the next statement after DO, in case of “success” the execution of the list of protected statements is repeated.

More detailed description of the input language is beyond the scope of this paper. To clarify our formal discussion of this language we will consider a sample program VISIT (Table I) for visiting all the nodes of the tree represented by three arrays SON, BROTH, FATHER. In each node program VISIT calls subroutine DOWN on descent and UP on ascent. In order to understand this program we have be familiar with the couple of procedures built into the input language. They are subroutine MOVE which writes character string into LOGICAL array, and Boolean function EQ which compares values of such array (left parameter) with character string and yields TRUE if they coincide.

```
PROGRAM VISIT
IMPLICIT INTEGER*2 (A–Z)
COMMON /TREE/ SON(100), BROTH(100), FATHER(100)
LOGICAL DIR(6)
I=1
CALL MOVE(”PASS”, DIR, 4)
IF (SON(I).NE.0) CALL MOVE(”DOWN”, DIR, 4)
NEXT=SON(I)
DO
    EQ( DIR,”DOWN”, 4) —>
    I=NEXT
    CALL DOWN(I)
    NEXT=SON(I)
    IF (NEXT.EQ.0) CALL MOVE( ”BRANCH”, DIR, 6)
    ¤
    EQ( DIR,”UP”, 2) —>
    I=FATHER(I)
    CALL UP(I)
    CALL MOVE(“BRANCH”, DIR, 6)
    ¤
    EQ( DIR, “BRANCH”, 6) —>
    IF
      BROTH(I).NE.0 —>
      NEXT=BROTH(I)
      CALL MOVE(“DOWN”, DIR, 4)
      ¤
      BROTH(I).EQ.0 .AND. FATHER(I).NE.0 —>
      CALL MOVE( “UP”, DIR, 2)
      ¤
      ELSE —>
      SKIP
    FI
OD
END
```

Table I. *A sample program on Dijkstra, a* FORTRAN EXTENDER *input language*

The FORTRAN EXTENDER supports two additional Fortran extensions which are very important for the design of efficient software: dynamic storage allocation and recursion. The dynamic storage allocation subsystem consists of the set of built-in subroutines. The subsystem is flexible enough to allow application of different algorithms of storage allocation and deallocation. Our experience showed that use of dynamic storage effected applications’ performance insignificantly.

The FORTRAN EXTENDER program is written in its own input language (with few Assembly routines). Its source text occupies about 1,000 lines, while object code requires about 70K byte.

12. EXECUTION MODE

In this mode, versions run under control of PW Run-Time Monitor. Excluding I/O support and switches from one terminal to the other, this Monitor registers exceptional situations during the execution of a version and intercepts an interrupt caused by this situation. We will consider some examples of exceptional situations.

Usually, such situations are caused by run-time errors, e.g., a data overflow, an attempt to update a protected storage location and so on. In all these cases the Run-Time Monitor intercepts an interrupt, processes it and passes control to the Debug Monitor to initiate a debug session for localization of the error.

Some situations are under special control of the Run-Time Monitor. One of them may happen when array indices exceed array bounds. Another situation may happen in case of disparity of formal and actual parameters of subroutines. SDE has static and run-time parameters check tools. While the static tool checks the correspondence of parameters in the design mode and during macrooperations, the dynamic tool conducts this check in the execution mode under Run-Time Monitor control. According to Fortran rules this situation is not necessarily erroneous. Thus, being turned on, this tool suspends the execution, displays inconsistent parameters and requests the designer’s permission to resume execution or to pass control to the Debug Monitor. It should be noted that in order to reduce possible performance degradation, all these checks can be executed very scrupulously and activated only for small pieces of version code. Nevertheless, even activation of parameters check for the entire version does not cause significant delays: it is implemented so efficiently that each subroutine check takes only as much time as the subroutine “call” itself.

One of the very important situations that can be sensed by the Run-Time Monitor is an infinite loop. It is necessary to protect CPU intensive programs from this error without claiming a low CPU time limit, because they can easily exceed this limit during an error-free run but execution will be erroneously canceled. It is especially important for unattended overnight runs or runs without on-screen output.

The main idea of the infinite loop protection is as follows (Fig. 10). Consider a program execution as running through labyrinth of subroutines (or even statements). We can approximately trace this path. After a few runs we can do it more precisely. Let us place “alarm clocks” in different points of this path and assume that all of them are “electrically wired and synchronized”.

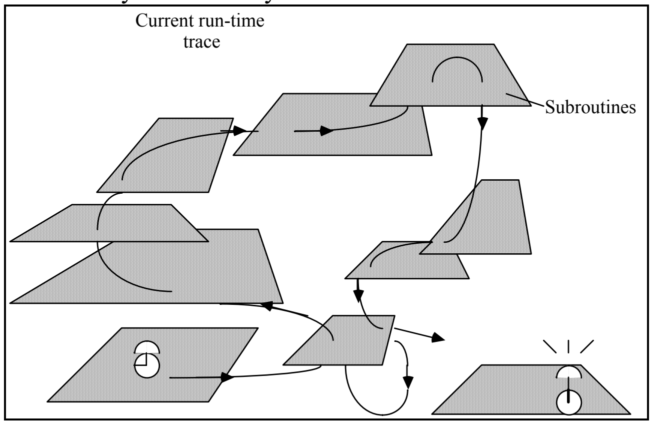


Fig. 10. *Run-Time Monitor: infinite loop protection*

Imagine that we can approximately evaluate CPU time necessary for the program to come from one clock to the next one. We need only the upper bound of this time and after a few runs we usually can do it easily. Now set up the first clock on a time interval, that is deliberately greater than the time required to reach the next clock. Imagine that during the execution, the program reached the next clock in time, i.e., before expiration of the limiting CPU time interval. Then we will “turn off the clock” to prevent ringing and reset this clock on a new time interval required to reach the next clock, and so on. Now assume that the program did not reach the next clock in time. The clock will “ring” and the program will be interrupted. It is very likely that the program delay was caused by an infinite loop along the path between the clocks (Fig. 10). Thus, the interrupt is justified.

The idea considered above is implemented in the Run-Time Monitor. For this purpose every version, in its kernel, has two subroutines. One of them allows setting the CPU timer to a certain time interval, the second intercepts and processes CPU timer interrupts. Interrupt processing is performed as follows: if the value of a logical variable BELL is “true”, then control is passed to the Debug Monitor, otherwise “true” is assigned to BELL, the CPU timer is reset to a new time interval and execution resumes. Thus, variable BELL located in COMMON /BELL/ BELL and, thus, being available everywhere, serves as a signal for two-way connection between the version and the subroutine for interrupt processing. During each time interval a version must “turn off the clock” at least once, i.e., assign “false” to BELL, otherwise the clock will “ring”, and the version will be interrupted. Subroutine DSPTIM is intended to perform this duty as well as to output and renew on screen a current version performance report and to accept input of user’s commands. Placing the statement:

```
IF (BELL) CALL DSPTIM
```

at certain check points in the source text allows us to detect an infinite loop and interrupt version execution if this version does not reach the next check point in time.

13. DEBUG MODE

Debug Monitor work procedures were considered in Sections 5 and 7. This Monitor receives control from the Run-Time Monitor if an exceptional situation is detected. Besides that, the control can be passed to the Debug Monitor at the check points predetermined by the designer. These points can be set up explicitly in the source text. They might be unconditional or have a condition in order to detect a run-time error. For example, we can explicitly check if array subscripts do not exceed array bounds. Moreover, we can link a check point with certain CPU timer readings and conduct checks only at specific time intervals.

Having received control, the Debug Monitor is able to display different kinds of information about the version including a current trace of subroutine calls, any program data and source texts. Let us consider a problem-oriented debugging subsystem, which considerably extends Monitor’s means of output and analysis of the information about current version state (Fig. 11). This subsystem allows a designer to write special debugging subroutines while preparing for testing and debugging. These subroutines can be activated by the Debug Monitor during the debug session. They might have parameters to be substituted by these version’s data structures or variables. This is performed by the Debug Monitor in a process of interaction with the designer just before starting debugging subroutine. It means the subroutine will process the current state of data as captured when the exception occurred. Debugging subroutines can solve two important tasks. First, they can convert and visualize version data structures into problem-oriented patterns (Fig. 11). This allows a designer to easily observe these structures and consequently easily understand their correctness (or find an error). Moreover, debugging subroutines can automatically generate "standard" (or expected under normal circumstances) values of these data, compare them with registered values and output diagnostics.

Actually, the Debug Monitor supports debugging in terms of the source text language, and even, in some sense, in terms of the problem being considered. At the same time this Monitor does not effect version performance.

14. CONCLUDING REMARKS

The SDE PW is implemented in its basic input language, the Dijkstra. The size of the source text is nearly 40,000 lines. Lower level communication and I/O subroutines are implemented in Assembly. SDE PW is an open system and is being extended permanently by adding new advanced tools. The system has been widely used for support of development of large-scale artificial intelligence research projects in several research institutions [9, 10, 13]. The applications designed, investigated, repeatedly redesigned, and maintained with PW support varied from 1,000 lines of source text to 70,000. The size of software research teams supported varied from one person to 16. Number of versions approached 120. Large numbers of applications and team size do not necessarily correspond to the upper limits of PW applicability. The experience of supported projects did not show these limits yet. Probably, for software teams of 100 members and more, it would be necessary to have a specific version control subsystem, e.g., implemented on DBMS, in order to manage groups of versions and their interactions.

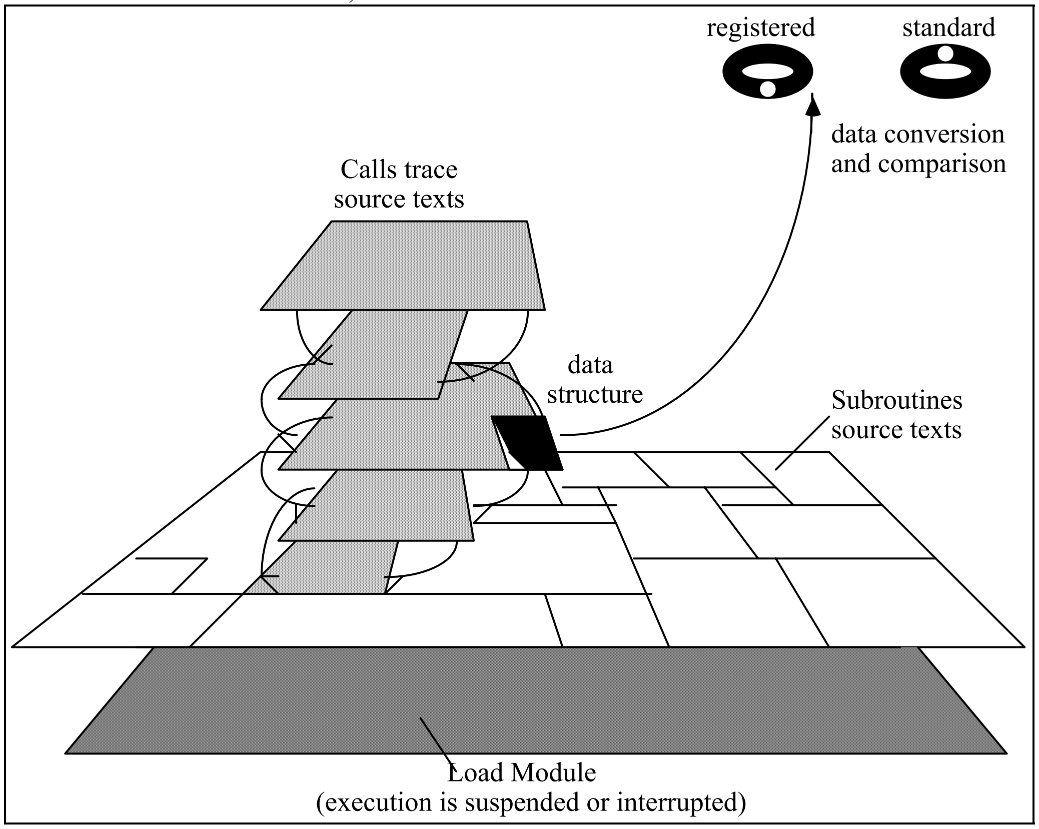


Fig. 11. *Debug Monitor: problem-oriented debug subsystem*

REFERENCES

[1] B. Boehm, “A Spiral Model for Software Development and Enhancement,” *IEEE Computer*, vol. 21, pp. 61-72, May 1988.

[2] G. Caldiera and V.R. Basili, “Identifying and Qualifying Reusable Software Components,” *IEEE Computer*, vol. 24, pp. 61-70, Feb. 1991.

[3] W.M. Osborne, and E.J. Chikofsky, “Fitting Pieces to the Maintenance Puzzle,” *IEEE Software*, pp. 10-11, Jan. 1990.

[4] Proc. Fourth ACM SIGSOFT Symposium on Software Development Environments, *ACM SIGSOFT Software Engineering Notes*, vol. 15, Dec. 1990.

[5] Special Section. Automating the Software Development Process: CASE in the 90’s, *Comm. of the ACM*, vol.35, pp. 27-89, Apr. 1992.

[6] N. Wirth, “Program Development by Stepwise Refinement,” *Comm. of the ACM*, vol. 14, pp. 221-227, Apr. 1971.

[7] O. Dahl, E. Dijkstra, and C. Hoare, *Structured Programming*, Academic Press, 1972.

[8] E. Yourdan and L. Constantine, *Structured Design*, Prentice-Hall, 1979.

[9] V.R. Mirniy, A.G. Roizner, M.V. Chudakov, B.M. Stilman, ”An Instrumental System for Support of Development and Debugging of Large-Scale Programs on IBM Fortran,” *Programming*, The USSR Academy of Sciences, May 1986, pp. 27-38, [in Russian].

[10] V.R. Mirniy, M.V. Chudakov, B.M. Stilman, “An Automated Programmers Workbench on Base of Integrated System of Program Versions,” in *Proc. of Int. Seminar on Software Engineering*, Moscow, 1988, pp. 59-65, [in Russian].

[11] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J.,1976.

[12] D. Gries, *The Science of Programming*, Springer-Verlag, 1983.

[13] B. Stilman, “A CASE System for Concurrent Development,” Dept. of Computer Science & Eng., Univ. of Colorado at Denver, Tech. Rep. TR–18, June 1992.