Precisely Annotated Hierarchical Pictures of Programs

David Lorge Parnas
Ashley Lawton
Software Engineering Research Group
DEPARTMENT OF COMPUTING AND SOFTWARE
McMaster University, Hamilton, Ontario, Canada L8S 4K1

1. On Pictures of Programs

In 1974, E. W. Dijkstra, criticising the use of pictures of programs, said, “Whenever someone draws a picture to explain a program, it is a sign that something is not understood”[2]. On hearing of this statement W. Bartussek responded with, “Yes, a picture is what you draw when you are trying to understand something or trying to help someone understand it”[1]. These two, superficially equivalent, statements demonstrate the broad range of opinions among programming experts about pictures of programs. The following opinions are, in our opinion, all accurate and relevant.

- Pictures clearly do help people to get an understanding of the basic structure of the program in their initial study of a program, or in finding their way through a large program to the sections that are of interest to them.
- Many diagrammatic representations of programs are *buzz-diagrams*, pictures that do not have a precise meaning. When dealing with programs, lack of precision can be dangerous and expensive.
- Diagrams can become so complex that they no longer convey the structure of the program to the reader.
- Pictures do not provide a way of summarising the behaviour of a software component and without a precise summary of the meaning of the boxes, the picture will have little meaning.

The importance of pictures was widely recognised in the early days of programming. Most early programming courses and textbooks advised drawing a flowchart before starting to write the actual “code”. The picture allowed programmers to plan a program rather than simply write code in the order that they think of it, or in the order that it will be executed. In the early days of programming, preparing the flowchart was considered the “design stage”.

Flowcharts are no longer as popular as they once were. Many textbooks now favour variations of programming by stepwise refinement [21], which is discussed further in Section 2. However, flowcharts are still found in some recent textbooks and discussions of algorithms. A new book, [18], provides a nice survey of both historical and current work.

Flowcharts of large programs can extend over hundreds of pages. In our experience these diagrams, especially those produced by flowcharting tools, are not very useful. Not only is it difficult to get an overview when the charts are so large, it is difficult to find the relevant details. Traditional flowcharts lack the hierarchical structure that is so valuable when studying any complex system. Someone charged with understanding a program written by others will want an overview of the structure and the ability to “zoom” in on the sections of interest. Conventional flowcharts do not support this type of use. Sometimes those who prepare flowcharts replace large parts of the flowchart with a labelled box, but the label only names the program. As it is not a

precise description of the program's function, the labels can be misleading. The top level pictures are imprecise buzz-diagrams, while the detailed pictures remain complex and hard to follow.

2. Programming by Stepwise refinement

For many program developers, the practice of drawing flowcharts has been replaced by the process known as "programming by stepwise refinement" in which a program is developed "top down", that is by writing an outline of the program and then filling in the details. The refinement process consists of a sequence of steps; in each refinement step, brief program names are replaced by more detailed descriptions until the text consists completely of executable code. Good illustrations of this idea can be found in [3, 21]. Stepwise refinement is undoubtedly useful during program development, but the structure becomes less visible with each refinement step. The final program is likely to be correct and well structured, but there is no record of the refinement steps in the code. Moreover, in the earlier steps, the function of the unrefined components is not precisely specified. consequently, refinement isn't necessarily helpful to those who "maintain[6]" old programs. Programs written by stepwise refinement have a hidden structure. Even worse, as the program is changed, that structure is destroyed [15].To help those who must understand the program after it has been written, we need documentation that provides a lasting record of the program structure.

3. The "display" method of program documentation

The *display method* [9], is a method of documenting programs that could have been produced by stepwise refinement. The method is based on the observation that while large programs must be examined in small pieces, the connections between those small pieces can be subtle and must be as precisely documented as the pieces themselves. In the display method, the programs, and their descriptions/specifications, are organised into a set of *displays*; each containing (1) the specification for that program, (2) the program text, and (3) the specifications for all[7] of the other programs invoked within that program. The specifications summarise program behaviour using a straightforward generalisation of Harlan Mills' "program function" [8, 13]. Tabular mathematical expressions, [11, 5], are used to make the representation of these functions easier to read.

Each display is *complete* in the sense that it can be understood and verified without looking at any other displays. The set of displays is *complete* if every[8] specification that appears in part (3) of one display, appears as part (1) of another display. A complete set of complete displays is a complete description of the program. If all displays in a complete description are correct, the program is correct. Programs that have been documented in this way can be systematically inspected and verified because the job of checking for correctness has been reduced to the mechanical job of checking for completeness, and a set of small jobs - namely checking individual displays. A significant practical use of this approach is described in [10, 12].

While we have found the display method useful in our work at McMaster, and in inspecting critical pieces of software, our experience has also revealed limitations. Although displays work well for documenting a program's components, they do not provide an overview of the program's structure. The reviewers may understand how each individual program segment functions, but still lack understanding of how all the pieces fit together and interact. The program that is used as an illustration in this paper is an excellent example of this point. The original documentation,

---

[6] Here, we follow terminology customary in the software industry and use "maintenance" as a euphemism for correcting errors and updating the functionality of programs that are in use.

[7] More precisely, specifications that would appear in more than one display are placed in a specialised dictionary (*lexicon*), to avoid duplication and inconsistency.

[8] In practice, we need not provide displays for built-in or primitive programs, programs whose availability is assumed by the program developers.

approximately 40 individual displays [16], was detailed, precise, and accurate; however, readers found it difficult to gain an understanding of the overall program structure from the displays. The original displays as Appendix A of this report so that interested readers can compare them with the diagrams we present.

4. Program Blueprints

This paper presents an approach to diagramming programs that incorporates the best features of the display method. It is designed to overcome the problems with pictures that were noted above.

Our approach to these problems was inspired by other areas of engineering, where hierarchically structured pictures are combined with precise mathematical descriptions of the components to yield documents that provide both an overview and precise details. In such diverse area as building construction and electrical circuitry, we find engineers using schematic diagrams or blueprints. These provide clear views of the structure of the product, but they also serve as an index into detailed technical descriptions. We illustrate a set of diagramming conventions that we call *software blueprints* which unites the idea of using pictures, with the accuracy and precision of the display method to allow a programmer to see both the overall structure and the details of a program.

At first glance, blueprints look much like flow charts. However, they have properties not shared by flow charts. The principal distinguishing characteristics are:

- All arrows flow from left to right - no “arrowheads” are needed.
- Loops are represented explicitly rather than implicitly by “backward” arrows,
- Boxes that represent a program contain precise summaries of that program’s behaviour,
- Like displays, each blueprint can be checked without referring to other blueprints.

Blueprints are based on the control constructs of a pseudo-code [13,14] based on Dijkstra’s guarded commands [4]. The control constructs are both simple and general. Using them, one can describe all well-structured algorithms. The diagramming conventions are based on the observation that there are only 4 ways to combine sequential program components to get larger sequential programs, invocation (possibly recursive), sequential composition, conditional composition, and iteration.

Invocation and sequential composition are represented exactly as in traditional flow charts except that a precise specification for the invoked program appears in the boxes when space permitsIf program A returns a value in an anonymous variable, that value can be identified by the symbol “#”.

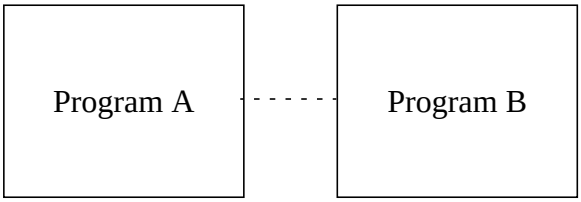


FIGURE 1 Sequential Invocation of Program B after Program A

Conditional composition is modelled on Dijkstra's guarded commands. Each alternative is represented by a path on the diagram. The "guarding condition" is written above the arc representing the alternative Note that the guards in the diagrams should be "pure" predicates, i.e.

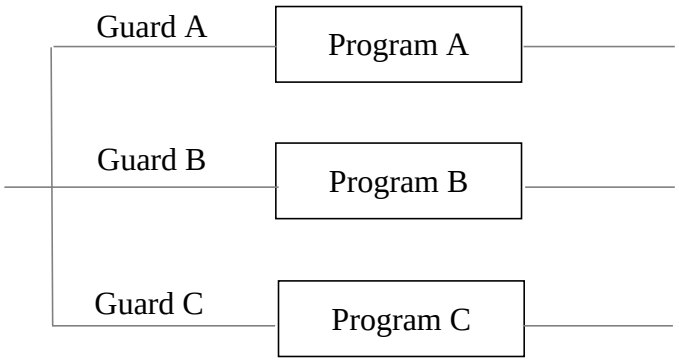


FIGURE 2 Conditional composition of Programs A, B, and C.

they should not change the state. If the actual code does have side effects, these should be shown as a separate box,

sequentially composed with the "guarded program". Guards with side-effects when they evaluate to ***false*** must have those side effects represented by boxes before the conditional composition. In general the "folk-wisdom" that side effects in boolean expressions are a bad idea is confirmed by the complexity of the diagrams that represent this accurately. The guards will be evaluated in some unknown order and one of the branches with a ***true*** guard will be selected. At least one guard should always be ***true*** otherwise the program will abort since none of the alternatives is valid.

Iteration is represented by heavy black boxes. The loop body, which can have more than one exit, is contained within the box. There are two kinds of exits from the body, those that lead to further iteration and those that exit from the loop. The exits that will lead to further iteration end at the box border and are marked with a GO (☛) symbol, while those that will exit the box are identified by a STOP (●) symbol[9].

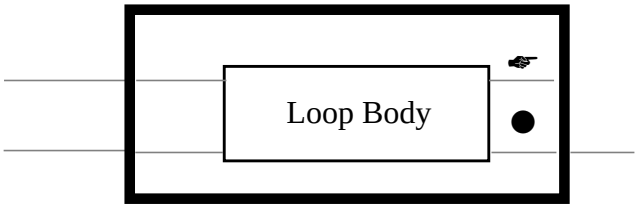


FIGURE 3 A typical loop

[9] In fact, the semantics [14] permits these symbols, which represent programs that set a control flag, to appear anywhere in the body of the loop that is not within an inner loop. The exit symbol is the last such program to be executed before the exit of the body is reached. Programs that take advantage of this generality are usually harder to understand than those where a decision can be unambiguously associated with each branch.

Note that the loop may have arbitrarily many entrances and exists. The loop body may be shown within the “box” or the box may contain a specification for the loop body.

Because the diagramming conventions are based on very general constructs rather than a specific programming language, they can be used for programs written in any imperative language. All of the common programming constructs can be shown to be special cases of those used in these pictures. A formal definition of the constructs can be found in [14].

5. Blueprints: A General Overview

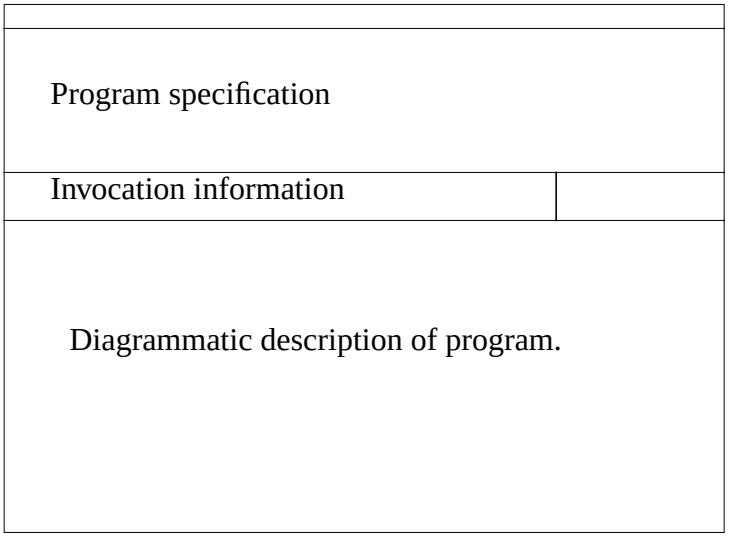


FIGURE 4

The picture above represents the basic layout of a software blueprint. The top section will contain the specification for the program described in the blueprint, the middle section will contain information about invoking the program, and the bottom section contains a diagrammatic description of the program. Each diagram is kept simple by replacing complex programs with their specifications. Where space permits, the box that represents a program contains its specification; in other cases the specification is nearby. As explained in the discussion of displays, if each of the blueprints needed for the program are correct, the whole program itself will be correct. A more complete discussion of completeness and correctness is found in [9, 20].

The individual blueprints are linked by the specifications. The specification found where the program is invoked appears again on the page that diagrams that program. The following picture illustrates the linking. In this example, spec 2 is repeated[10] in blueprint 2, spec 3 in blueprint 3, etc. Each dotted line represents a chance to “zoom” in for a closer look at a part of the program.

[10] In fact, supporting tools would not repeat the specification; only one copy would be stored but each display would include a copy when printed or displayed. See [20].

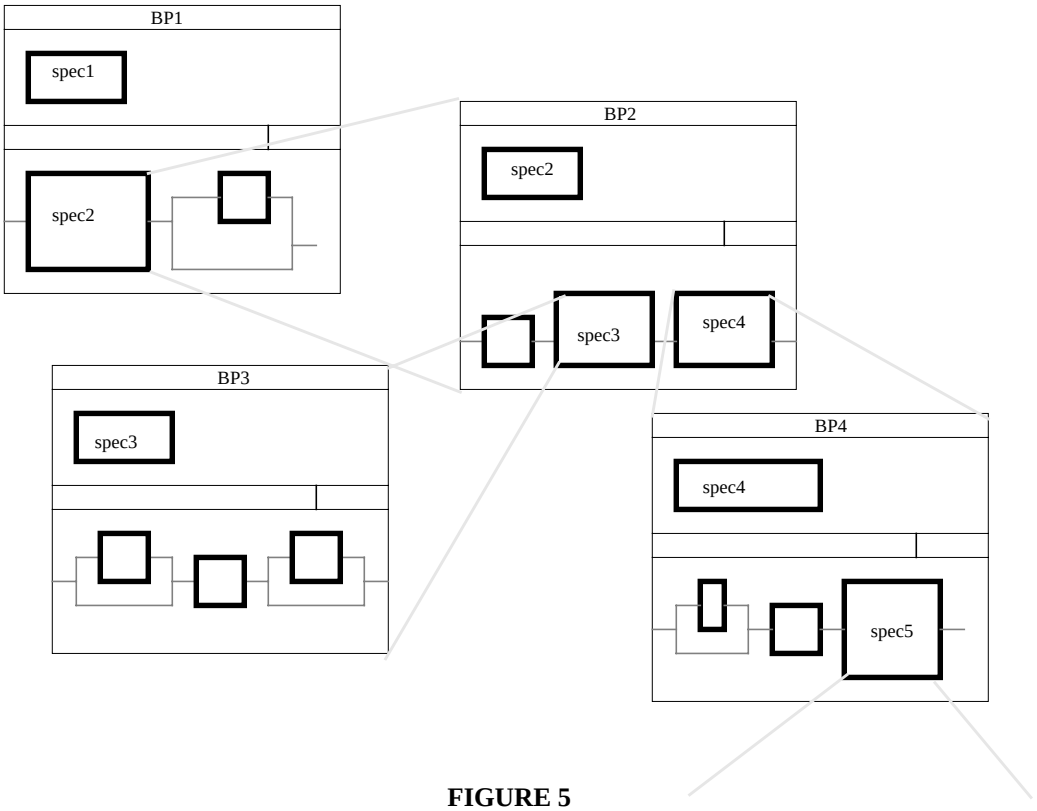


FIGURE 5

6. The Blueprint: A Detailed Description

Below is an example of a blueprint for a C program. The specification notation used in these examples is explained in [9]. While blueprints can use any specification notation, one must be chosen for examples. The examples are not from a “toy” program (one developed to serve as an example) but are taken from one of the tools developed to support the use of tabular notations. These tools are described in more detail in [16, 17, 6, 7].

FIGURE 6

❶ ***Blueprint 2.7***

2.7 Specification ❷

❹ *From Blueprint 2.3*

void IsNormalHeader (Expn expnh) ❸

$\mathbf{R}_{\text{IsNormalHeader}}(,)=$

	Is_Normal_header (Normal, expnh)	¬*Is_Normal_header* (Normal, expnh)
status' |	status'=Success	¬(status'=Success)

∧NC(expnh)

①

2.7 Invocation ❺ `IsNormalHeader(expnh)`

KEY ❻ `expnh`: type `Expn`

2.7 Dadgram ❼

Declarations ❽

Type	Name
int	i1, dim1
Token	status

④

❾

$\mathbf{R}_{\text{IsNumGridHeaderMatch}}(,)=$

	$^{\text{numgrid}}\text{shape}_{\text{expnh}}$ = table_dim$_{\text{expnh}}$	¬($^{\text{numgrid}}\text{shape}_{\text{expnh}}$ = table_dim$_{\text{expnh}}$)
status' |	status'=Success	¬(status'=Success)
dim1 |	dim1=table_dim$_{\text{expnh}}$	true

∧NC(expnh)

On Blueprint 2.8 ③

❿

❶❶ status = GetErrTH(); i1 = 1

(i1 <= dim1) && (status==Success)

$\mathbf{R}_{\text{IsConditionHeader}}(,)=$

	$(\forall j,(1 \le j \le {}^{\text{len}}\text{shape}_{\text{expnh}}, i1,1) \rightarrow$ *Is_Condition*(expntypecell$_{\text{expnh, i1 ,j}}$)	
	true	*false*
status' |	status'=Success	¬(status'=Success)

∧NC(expnh, i1)

❶❷ *On Blueprint 2.12*

status = GetErrTH(); i1=i1+1

❶❸

!((i1 <= dim1) && (status==Success)) ②

❶ A number uniquely identifying the current blueprint.

❷ The mathematical specification for the program that appears on this blueprint.

❸ The formal parameter list. This indicates the type and name of all formal parameters used as well as the type and name of the program.

❹ Where this program is invoked - blueprint identifiers for all programs that invoke this program.

❺ Text that invokes this program. This line comes directly from the program code.

❻ The types of the actual parameters. This can be used to check that the actual parameters are of the same type as their corresponding formal parameters.

❼ The pictorial representation of the program code that implements the specification in ❷.

❽ The declaration of local variables. If the scope is smaller than the whole blueprint, dotted lines define the scope. This is illustrated in Figure 7.

❾ The specification of another program that will be invoked at this point.

❿ The blueprint number for the program being invoked.

❶❶ Program instructions taken directly from the program code.

❶❷ Program guards. When executed, the program will select a branch whose “guard” evaluates to TRUE. If more than one branch can be selected, selection is non-deterministic.

❶❸ The iteration box. This thick black box encloses program code which may be repeated. The STOP (●) or GO (☞) symbols identify paths that lead to termination or continuation of the iteration.

Line Styles

① The border and lines that separate different sections of the blueprint are solid black lines of a .5 point thickness.

② The connecting lines that join the program pieces together are hashed black lines of a .5 point thickness.

③ The boxes that contain specifications and program instructions and declarations are solid black lines of a 2 point thickness.

④ The iteration boxes are solid black lines of a 4 point thickness.

FIGURE 7

Blueprint 1.1

1.1 Specification

void swtch (int a[ ], int b[ ], int size)

$\mathbf{R}_{swtch}(,)=$

$\sum_{i=0}^{size} a[i] > \sum_{i=0}^{size} b[i]$	$\neg \left(\sum_{i=0}^{size} a[i] > \sum_{i=0}^{size} b[i] \right)$
$\forall i, (0 \le i < size) \Rightarrow$ $(`a[i]=b'[i] \wedge `b[i]=a'[i])$	NC(a,b)

$\wedge$NC(size)

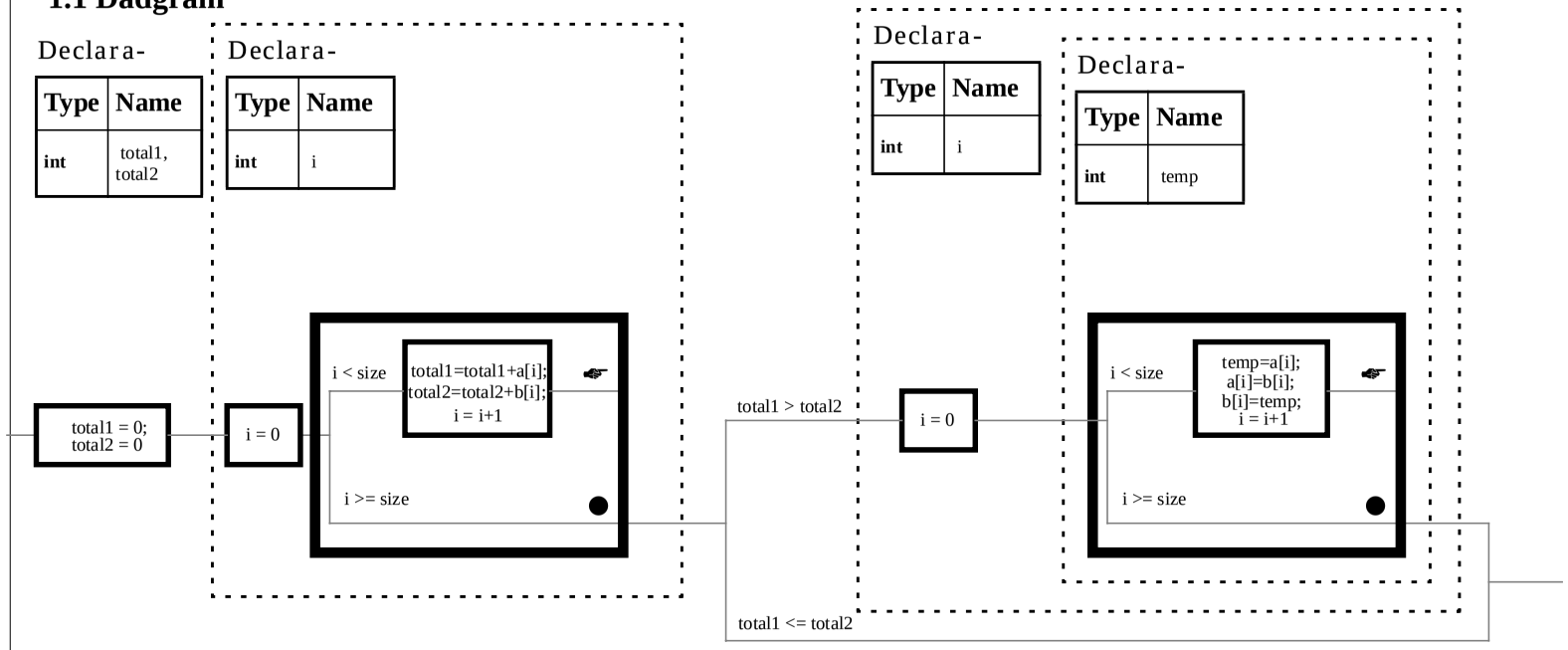
1.1 Invocation

```
swtch(a, b, size)
```

KEY

a: type `int`
b: type `int`
`size`: type `int`

1.1 Dadgram



Components Not Yet Covered

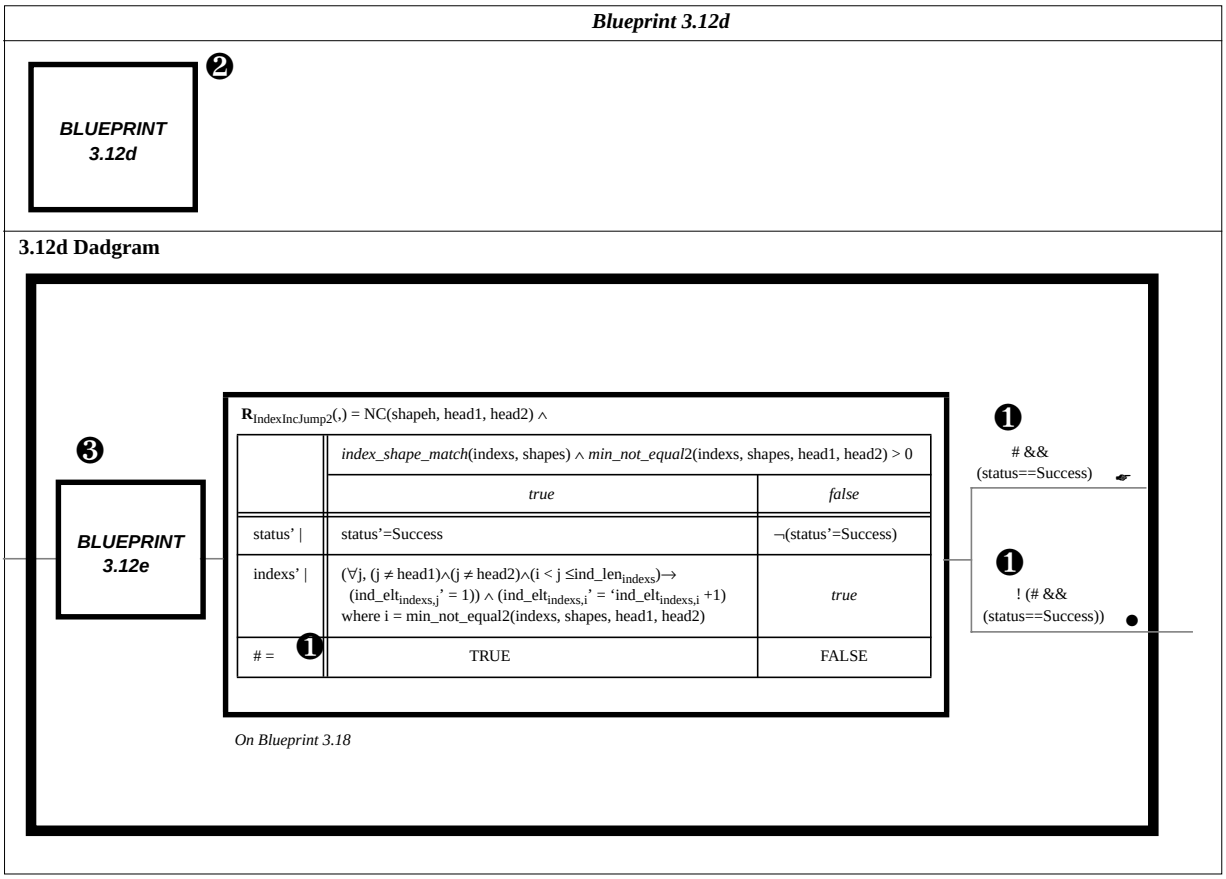


FIGURE 8

❶ The # symbol is used to denote the values returned by functions.

❷ Unfortunately it can not be guaranteed that all program blueprints will fit onto a single page. This is why there is a box stating 'BLUEPRINT 3.12d' where there is normally a specification. The text in the box indicates that this is not a complete blueprint but belongs to blueprint 3.12, and that it is the 4th page.

❸ Since this is not the last page of blueprint 3.12, reference must be made to another subsequent part, 3.12e. Everything which appears on blueprint 3.12e should actually appear in the box labeled BLUEPRINT 3.12e and so on all the way back to blueprint 3.12. This may be more easily understood with the aid of the following diagram:

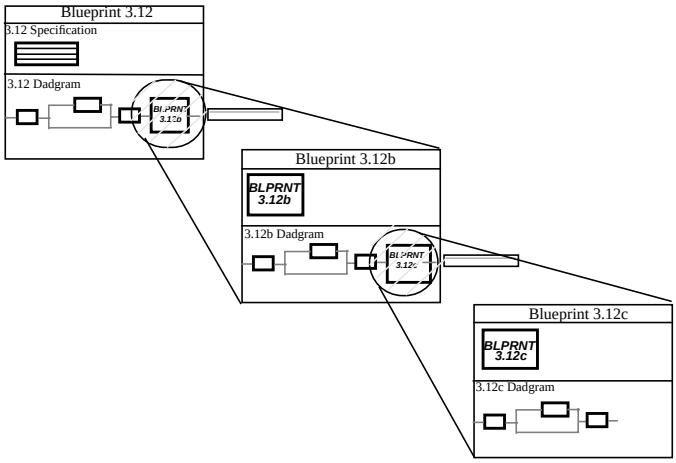


FIGURE 9

There is no invocation line on this blueprint because this is not a complete blueprint, but only one page of a series and that the invocation appears on the first page in the series.

These are the guidelines and conventions which have been followed in the creation process of these blueprints. The programming language used in the diagrams is the same language as is used to write the program which the blueprints represent.

7. Conclusions and future work

The process of making blueprints for a fairly large programme leads to improvements in the programs. Whenever you make structure explicit, you will find ways to improve it. It is vital to keep the line-styles, box-styles, etc. consistent because they provide visual clues for people trying to understand the program.

While our original idea was to get every diagram to fit on one page, our final result was more pragmatic. Programs may be large and still simple, i.e. not difficult to understand. These need not be decomposed to the point where every picture fits on one page.

The formal specification notation often seems to take up too much room. We continue to look for ways to reduce the “bulk” in these notations without reducing either precision or clarity.

The blueprinting process does not have to be used as a reverse engineering tool, but would also be beneficial as a design tool. If the blueprinting method is used as a design tool then not only would it provide useful documentation but it would also ensure that the program would be well structured. There is no reason why blueprints could not be input to a program generator.

We would like to have a tool used for viewing blueprints on a computer instead of on paper. In this way one wouldn’t need to consult another piece of paper when confronted with a box indicating that the blueprint is continued. Instead, an individual could simply click onto the box and it would be enlarged, showing what would have been on the separate page. This would greatly simplify the viewing process. The tool could also do for blueprints, what DMS [20] does for displays. DMS checks for completeness and notifies reviewers when something has changed that might invalidate their review.

Of course, generating blueprints by hand, which is how the blueprints contained in Appendix B have been created, is a time consuming, somewhat trivial process which can be error prone. For this reason, it would be nice to have a tool that would generate program blueprints from program code. We suspect that such a tool would have to be interactive, i.e. that the process cannot be fully automated.

We conclude that it is time to stop thinking of pictures and formal methods as alternatives. First, pictures can be formal. Our diagrams do have a formal meaning and could be translated to non-pictorial representations. Further, the pictures and the more conventional formal notations complement each other producing documentation that is better than could be produced with either method alone.

As with any form of program specification/documentation, there is always a problem when dealing with parameter passing. The semantics of parameter passing in most programming languages is sufficiently messy that it has defeated most attempts at formalisation. Since this proposal is about diagrams, not language design, or program specification, we choose to document each invocation rather than document the procedure declaration. We created a new blueprint for each call to a program when different parameters are passed to it. Further thoughts on this problem can be found in [19]. Probably, the only clean solution will require a new approach in the programming language itself.

8. References

1 W. Bartussek, personal communication.

2 E.W. Dijkstra, remark made at meeting of IFIP working group w.g.2.3

3 Dijkstra, E.W. “Notes on Structured Programming” Structured Programming, Dahl, O-J., Dijkstra, E.W., Hoare, C.A.R. (Ed.), Academic Press, 1972.

4 Dijkstra, E.W., “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”, Communications of the ACM, August 1975, pp. 453-457.

5 Janicki, R., Parnas, D.L., Zucker, J., “*Tabular Representations in Relational Documents*”, in “Relational Methods in Computer Science”, Chapter 12, Ed. C. Brink and C. Schmidt. Springer-Verlag, pp. 184-196, 1997, ISBN 3-211-82971-7.

6 McMaster University Software Engineering Research Group, “Table Tool System Developer’s Guide” CRL Report 339, McMaster University, CRL (Communications Research Laboratory), TRIO, January 1997.

7 McMaster University Software Engineering Research Group, “Appendices to the Table Tool System Developer’s Guide” CRL Report 340, McMaster University, CRL (Communications Research Laboratory), TRIO, January 1997.

8 Mills, H. D. “The New Math of Computer Programming” Communications of the ACM, Vol. 18, No. 1, pp. 43-48, January 1975.

9 Parnas, D.L., Madey, J., Iglewski, M., “Precise Documentation of Well-Structured Programs”, *IEEE Transactions on Software Engineering*, Vol. 20, No.12, December 1994, pp. 948 - 976.

10 Parnas D.L., Asmis, G.J.K., Madey, J., “Assessment of Safety-Critical Software in Nuclear Power Plants”, Nuclear Safety, Vol. 32, No. 2, April-June 1991, pp. 189-198.

11 Parnas, D.L., "*Tabular Representation of Relations*", CRL Report 260, McMaster University, Communications Research Laboratory, TRIO (Telecommunications Research Institute of Ontario), October 1992, 17 Pgs.

12 Parnas D.L. "Inspection of Safety Critical Software using Function Tables", Proceedings of IFIP World Congress 1994, Volume III" August 1994, pp. 270 - 277.

13 Parnas D.L. "A Generalized Control Structure and Its Formal Definition" Communications of the ACM, Vol. 26, No. 8, pp. 572-581, August 1983

14 Parnas D.L., Wadge, W.W., "Less Restrictive Constructs for Structured Programs", Technical Report 86-186, Queen's, CIS, Kingston, Ontario, Canada, October 1986, 16 pp.

15 Parnas, D.L., "Software Aging", in *Proceedings of the 16th International Conference on Software Engineering*, Sorrento Italy, May 16 - 21 1994, IEEE Press, pp. 279 - 287

16 Shen H., "Implementation of Table Inversion Algorithms", CRL Report No. 315, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, ON, Canada, December 1995, 96 pgs.

17 Shen H., Zucker J.I., Parnas, D.L., "Table Transformation Tools: Why and How", Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS '96), published by IEEE and NIST, Gaithersburg, MD., June 1996, pp. 3-11.

18 Stasko, John T., John B. Domingue, Marc H. Brown, Blaine A. Price, editors. Software Visualization: Programming as a Multimedia Experience. Cambridge, MA: The MIT Press, (forthcoming 1998).

19 Iglewski, M., Madey, J., Parnas, D.L., Kelly, P. "Documentation Paradigms (A Progress Report)", CRL Report 270, McMaster University, CRL (Communications Research Laboratory), TRIO (Telecommunications Research Institute of Ontario), July 1993,45 pgs.

20 Wang, Yali, "*Display Management System*", *A tool to support the Display Method*. CRL Report 297, McMaster University, Communications Research Laboratory, TRIO (Telecommunications Research Institute of Ontario), February 1995.

21 Wirth, N., "Program Development by Stepwise Refinement", Communications of the ACM, Vol. 14, No. 4, April 1971, pp. 221-227.

Appendix A: Blueprints for Table Inversion Program

When comparing the blueprints to the displays, it will become obvious that there are more blueprints than displays. This disparity arises from the fact that separate blueprints were generated for separate calls to the same programs, as already mentioned. This was not the case, however, for the displays. Therefore, blueprints 2.18 to 2.23 and 3.22 to 3.26, inclusive, do not have a direct correspondence with any displays.

Appendix B: Displays for Table Inversion Program[6]

[6] This section was extracted from: Shen H., "*Implementation of Table Inversion Algorithms*", *CRL Report No. 315*, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, ON, Canada, December 1995, 96 pgs.