TOOLS FOR LARGE-SCALE SOFTWARE ENGINEERING

Robert G. Babb II
Oregon Graduate Center

Abstract

The goal of this project is the establishment of a support environment for programming via direct implementation of data flow diagrams. Programs are viewed as being made up of data coupled systems of data-activated processing elements. Processing elements are linked together by data paths to define processing subtasks and interaction patterns. System data flow diagrams are used to graphically display data linkages. A coherent hierarchy of system diagrams defines how complex systems are composed from successively simpler systems. The lowest level system actions are specified by more or less conventional programs.

The system has three primary software tools: an interactive graphics-based designer interface for data flow diagrams; a system design tracker that allows simultaneous, multiperson design changes; and data flow diagram "compiler" that creates data-activated program drivers.

As a result of this research we hope to greatly enhance our ability to design large systems of programs via:

1) complete integration of software specification, prototyping, and production
2) a technique for formal verification of consistency between "as specified" system requirements and "as built" system designs
3) cost effective re-use of program modules and system data flow patterns
4) a practical paradigm for the design of complex systems of cooperating sequential processes.

1. Problems with Engineering Large-Scale Software

This research project is aimed at improving our abilities to develop large systems by building a set of tools to integrate software specification, design, and implementation. A variety of definitions for "large-scale" software have been proposed[1]. At a minimum, systems developed and maintained by more than one person usually exhibit problems such as those addressed by this proposal.

Recent research suggests that human abilities to design software systems can be enhanced through use of graphic design notations[2] [3], although there seems to be no advantage to the use of traditional detailed flowcharts for programming[4]. Hardware design engineers have relied heavily on the use of graphic design notations such as block, circuit, and timing diagrams for some time. However, the use of two dimensional program design notations by software engineers is not yet common. While diagram notations have been used in documenting requirements, their integrated use for software system specification, design and implementation is a novel feature of this research.

The process of developing software today has little in common with the methods used in traditional fields of engineering. As both Dijkstra[5] and Hoare[6] have observed, programming resembles a craft, in which each new project is started from scratch, with little or no re-use of previously engineered components. As a result, most software in existence today does not have the basic qualities of a well engineered product[7]:

- reliability
- maintainability
- predictability of development cost and schedule

In an attempt to overcome these problems, the establishment of the field of "software engineering" was proposed at two conferences held a little over a decade ago[8] [9]. Initial efforts in this area concentrated on improving methods for "programming-in-the-small." In particular, the search for more disciplined programming methods concentrated initially on the control structures of imperative languages, leading to the development and widespread acceptance of the techniques of "structured programming." Use of structured programming techniques has enhanced our ability to construct reliable programs, especially when coupled with proof of correctness considerations[10].

As early as 1975, however, it was recognized that designing large (systems of) programs is a fundamentally different problem from those addressed by structured programming and proof of correctness techniques[11] [12]. We will use the term *system* to refer to such large-scale programs. Numerous techniques and tools have been proposed for describing and specifying requirements for software systems[13]. Unfortunately, due primarily to the limited scope and ad hoc nature of these methods, there is little coherence between current requirements specification and software development techniques. The difficulties that result include[14]:

- ambiguous, even self contradictory, requirements specifications
- poor communication between users and developers
- lack of traceability between requirements and designs
- lack of up-to-date "as built" system documentation

Many of these problems with "programming-in-the-large" are related to the lack of visibility of evolving software designs and to the difficulty of communication among system developers. Especially in large (i.e. multiperson) projects, it is usually impossible to obtain reliable, up-to-date information about the "as partially built" system design.

A more fundamental difficulty with most current approaches to system requirements specification and design is that they tend to *describe* requirements informally, rather than to *define* them formally. One exception is the so called operational approach to requirements definition, in which a model of a desired system is constructed that can be executed to simulate the interaction of the eventual system with its environment[15] [16]. These techniques are also related to the concept of *rapid prototyping* for evolving software designs[17].

Ongoing changes in computer architectures are making the the problems of large-scale software engineering even worse. The desire for increased speed, reliability, and distribution of information processing has led to employment of pipelined, parallel, and distributed system architectures, for which current techniques of designing software will likely prove increasingly inadequate. The desire to effectively utilize the processing power becoming available as a result of VLSI design techniques[18] is also making the solution of the large-scale programming problem more urgent. Interestingly enough, VLSI designers are also experiencing difficulties with increasing complexity of their designs and are investigating the application of software engineering techniques to aid VLSI design[19].

2. Systems of Data-Activated Programs

Programmers tend to put themselves *into* their programs, in tightly coupled control of the processing steps. However, even many "small" programming problems can be more easily solved in a loosely coupled *system* fashion. An analogy may be useful. If the control flow in an ordinary program resembles a person, a system runs more like a business organization. In businesses, paths for information flow (channels) become established. After a memo arrives in a person's in basket, the recipient will, at some future time, act on the information, perhaps by sending other memos. The net effect is that organizations conduct business in a loosely coupled, data driven fashion. No one person needs to know about and control everything that happens.

Similarly, in the proposed research, programs are viewed as being made up of data coupled systems of data-activated processing elements. Processing elements (**p** 's) are linked together by data paths (**d** 's) for the purpose of defining processing subtasks and interactions. System data flow diagrams are used to graphically display and define linkages between **p** 's and **d** 's. A coherent hierarchy of system diagrams defines how complex systems are composed from successively simpler systems. Processing elements not specified by system diagrams are specified by more or less conventional programs. The actions of these programs can include:

— decisions based on input data values
— calculating output data values
— reading (consuming) input data values
— writing (producing) output data values

All global system control flow considerations are handled implicitly by appropriate specification of local data flow control actions (consuming and producing data values).

A system can be in one of three states: *suspended*, *executing*, or *terminated*. Systems can start execution only if data is present on all inputs, and all output data paths are available for writing. Each time a system suspends, a check is made for "data flow progress." If data flow progress was not made (at least one input consumed, or one output produced), the system is terminated, and blocked from further execution.

As an example of how design would proceed using our approach, we present a system solution to the following text processing problem:

> Excess blanks (i.e., all but one in a series of blanks) are to be removed between the words in a file. The file contains a series of fixed length data records, but "words" can be of any length, and can extend across record boundaries. The resulting words are to be packed into a series of fixed length output records. Assume for concreteness that all input records are 80 characters long, and that all output records are to be 60 characters long.

The presentation is informal, with notations and semantics explained as part of the discussion. The language C[20] as implemented on UNIX† is used for the example.

Top level data flow diagrams are normally designed before lower level programs are written, although design can proceed bottom-up as well. Backtracking is frequently required when data flow or program design decisions are made at lower levels that have impact on the upper level data path design.

At the top level, as shown in Fig. 1., the "program" p0 (squeeze) has two input and two output data paths. Unique system tags (e.g. "p0") are used for unambiguous referencing of system processing elements. Normally, data paths also have an associated tag (d1,d2, ...). Outermost data paths, such as those shown in Fig. 1, typically do not have tags because they are assumed to be under the control of an underlying operating system, rather than of the data flow scheduler.†† An asterisk next to a data path indicates that its capacity is not

†TM, Bell Telephone Laboratories, Inc.

††Untagged data paths serve to document important side effects that are expected to occur as a result of

```
/*=================================================== squeeze  p0 ==*/
/*                                                        [top]   */
```

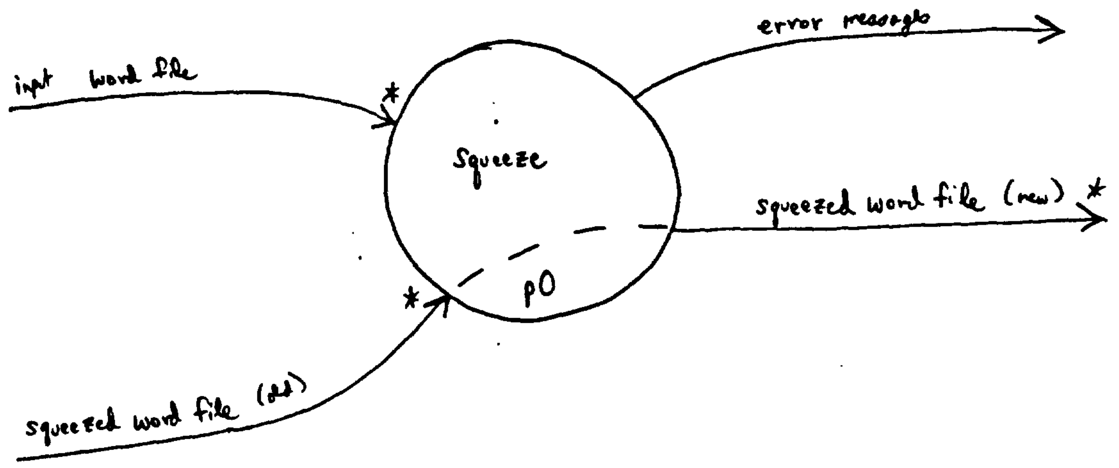

Fig. 1. Top-level system diagram for p0 (squeeze).

fixed (i.e., it behaves like a "file"). The dashed line through the circle indicates that the contents of the squeezed word file are updated as a result of running this system. It may seem odd that the primary output of this system is also shown as an input. However, this reflects that in general (under UNIX) writing to a file destroys any previously existing file of that name in the current directory.

Shown in Fig. 2 is a specification for p0 (squeeze) as a *system*. All **p** 's are defined as either programs or systems. Examples of program specifications are shown in Figs. 3,4,6,7,9,10, and 11 below. The input and output data paths on any **p** are constrained to match exactly the inputs and outputs as shown on the corresponding *context* diagram. Context diagram tags are shown within square brackets in the upper right corner of system diagrams. In this case, we are at the [top] level of the design. The solid lines through p4 connecting data path d1 with d2 and d3 with d4 indicate "read-only" access to shared data[†]. Note that the squeezed word file is shown consistently as being updated both here and in Fig. 1. Names written within quote marks below data paths correspond to data names in the associated C program code.

running a system. Even though these side effects are not under "data flow control," they generally reflect the purpose of the system!

[†]Shared means that only one copy of the data is stored in the system. Implementation of access control strategies to avoid read-read and read-write conflicts are not currently planned as part of the implementation system, but are the responsibility of the system designer.

```
/*=====:=====================================  squeeze  p0 ==*/
/*                                                  [top]  */
```

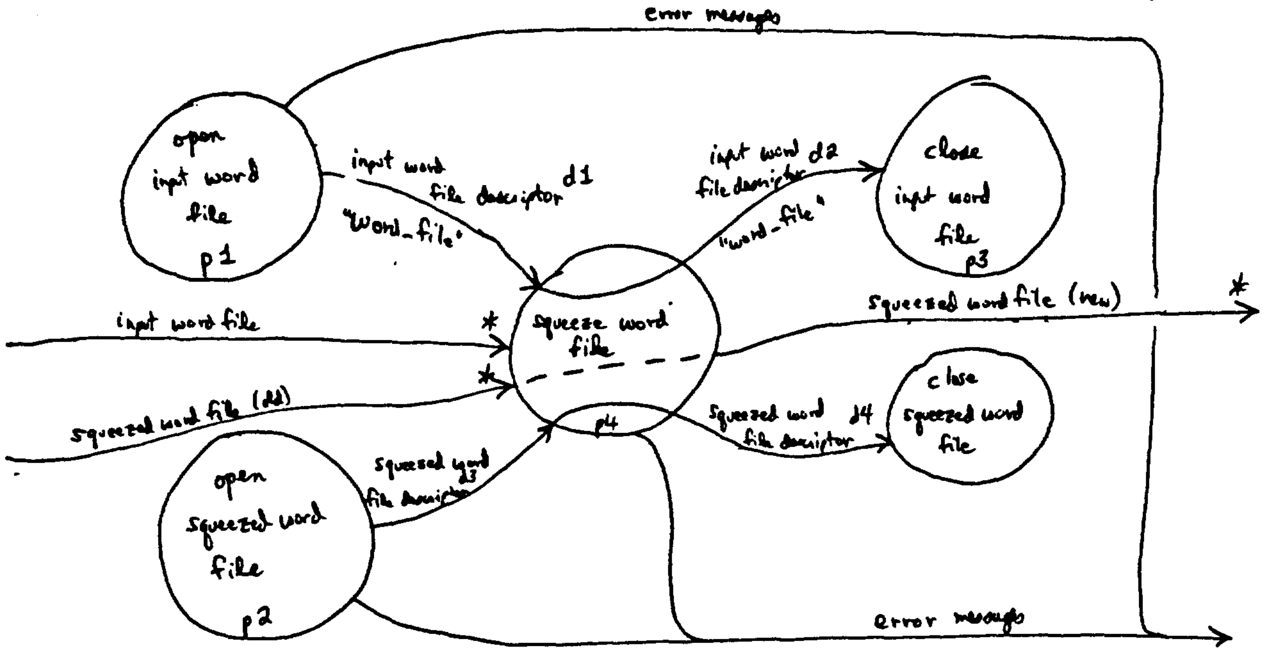

Fig 2. System specification for p0 (squeeze).

An example of a program specification is shown in Fig. 3 for p1 (open input word file). The language used for this example is C, although any language appropriate to a particular implementation environment can be used for this style of programming. Some parts of the program text, written in capital letters, are macro calls that abbreviate standard data flow control actions (BEGIN,END,SET,SUSPEND). The switch label s0 on line 6 is part of a built-in program state mechanism. Each program has available an initial state (s0) and

```
 1  /*:===========================  open  input  word  file   p1 ==*/
 2  /*                                                    [p0]   */
 3  p1(d1)
 4  struct { FILE *word_file } *d1;
 5  BEGIN(p1)
 6     s0: /* open */
 7         if((d1->word_file = fopen("record_file","r") == NULL)
 8             printf("p1: open error\n");
 9         else
10             SET(d1);
11     SUSPEND;
12  END(p1)
```

Fig. 3. Program specification for p1 (open input word file).

a number of additional states for use in saving local state information between data driven invocations. Data definitions (e.g. Fig. 3, Line 4) are placed inside C "struct" declarations with **d** -tag names, so that all references to data by programs will be tied directly to data paths as shown on system data flow diagrams. Read/write vs. read-only access priviledges are distinguished using the pointer (e.g. Fig. 3, Line 7) and dot (e.g. Fig. 4, Line 7) structure referencing mechanisms provided in C.

The BEGIN and END macros expand into calls to built-in scheduler functions that implement tests for data driven program and system activation based on the current "data state" of the system. Data paths (**d**'s) can be in one of three states: clear, set, or eof. When a program has finished computing the value on an output data path, a SET command (e.g. Fig. 3, Line 10) makes the value available "downstream." When a program is finished with an input data item, a CLEAR command (e.g. Fig. 4, Line 8) makes the data path available for writing "upstream." The program for p2 (open squeezed word file) is isomorphic to the program for p1†.

Shown in Fig. 5 is a system specification for p4 (squeeze word file). The program p6 (read input word file record) is shown in Fig. 6. Here we see an example of ENDing a data path (Fig. 6., Line 14) to signal an "end of file" condition. End of file conditions are also useful for non-file-related activities. Also, notice the distinction between the two kinds of access to d5 (read/write) and d1 (read-only) on Line 8.

The program for p7 (write squeezed word file record) is shown in Fig. 7. Note that p7 tests for the end of file condition using a built-in data flow predicate function EOF on Line 7.

The programs p9 and p10 referenced on system p4 (Fig. 5) serve to initialize cursors (d7, d8) used for extracting characters from input records and filling output records.

```
 1  /*==========================  close input word file  p3 ==*/
 2  /*                                                    [p0]  */
 3  p3(d2)
 4  struct { FILE *word_file } d2;
 5  BEGIN(p3)
 6      s0: /* close */
 7          fclose(d2.word_file)
 8          CLEAR(d2);
 9      SUSPEND;
10  END(p3)
```

Fig. 4. Program specification for p3 (close input word file).

†We hope to identify and parameterize many such common programming tasks as a side-effect of the use of this set of software tools.

```
/*=============================================== squeeze word file  p4 ==*/
/*                                                                 [p0]  */
```

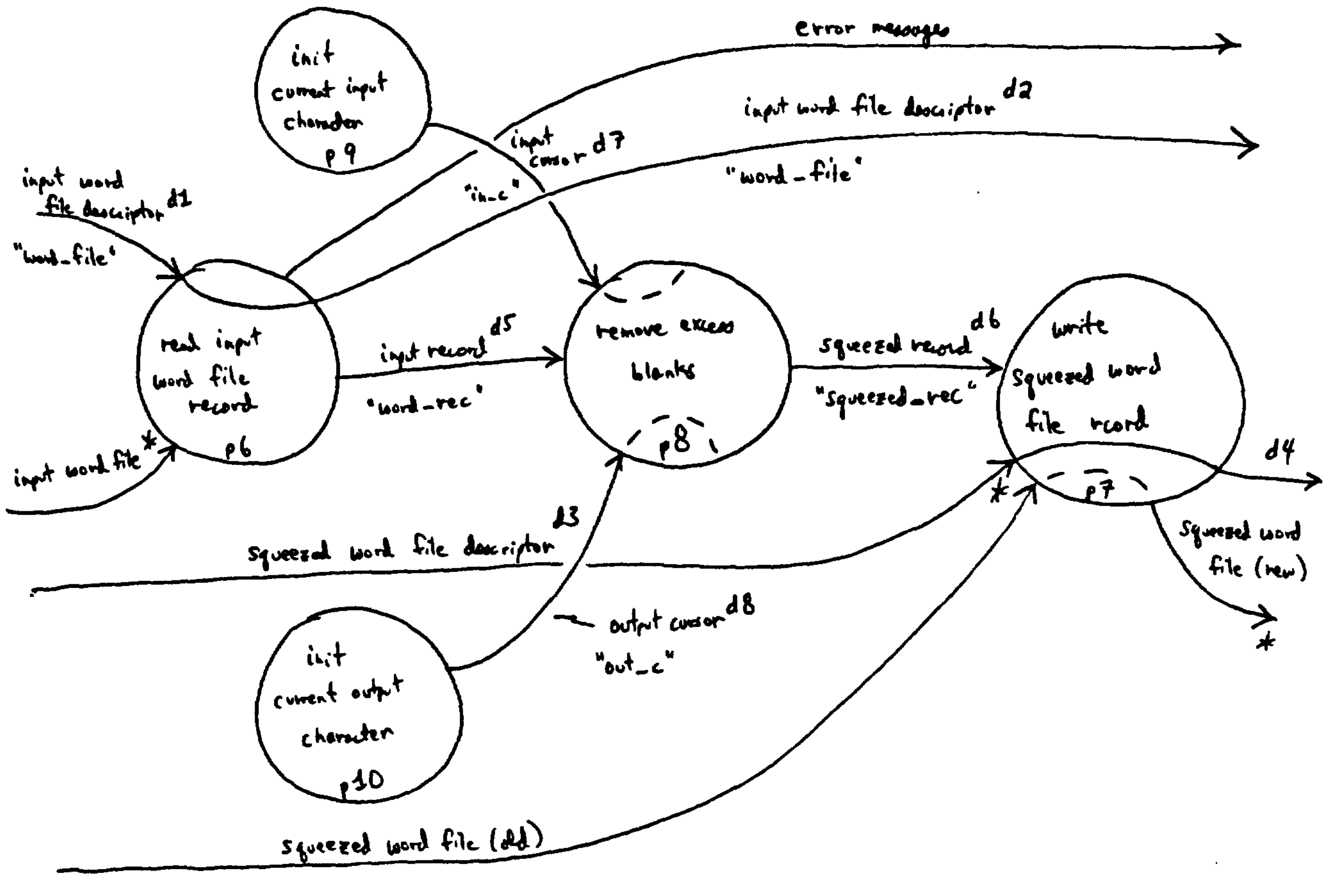

Fig. 5. System specification for p4 (squeeze word file).

```
/*==================== read input word file record  p6 ==*/
/*                                                  [p4]  */
p6(d1, d2, d5)
struct { FILE *word_file } d1, d2;
struct { char word_rec[80] } *d5;
BEGIN(p6)
   s0: /* read */
      if(fgets(d5->word_rec, 81, d1.word_file != NULL)
      {  if(strlen(d5->word_rec) != 80)
            printf("p6: error in input record length\n");
         else
            SET(d5); }
      else /* end of file */
      {  CLEAR(d1); SET(d2); END(d5); }
   SUSPEND;
END(p6);
```

Fig. 6. Program specification for p6 (read input word file record).

```
/*================== write squeezed word file record  p7 ==*/
/*                                                   [p4]  */
p7(d3,d4,d6)
struct { FILE *squeezed_file } d3, d4;
struct { char squeezed_rec[60] } d6;
BEGIN(p7)
   s0: /* write */
      if(EOF(d6))
      {  CLEAR(d3); CLEAR(d6); SET(d4); }
      else /* record */
      {  fputs(d6.squeezed_rec,d3.squeezed_file); CLEAR(d6); }
   SUSPEND;
END(p7);
```

Fig. 7. Program specification for p7 (write squeezed word file record).

Shown in Fig. 8 is a system specification for p8 (remove excess blanks). The program p11 (characterize input record) breaks up input records (d5) into characters (d9) with the aid of the input cursor (d7). The resulting stream of characters is filtered by p12 (de-blank) to remove all but one of series of blanks. The resulting stream of processed characters (d10) is assembled by p13 (assemble squeezed records) into squeezed records (d6) with the aid of the output cursor (d8). The corresponding C code for p11-p13 is shown in Figs. 9-

```
/*================================== remove excess blanks p8 ==*/
/*                                                     [p4]  */
```

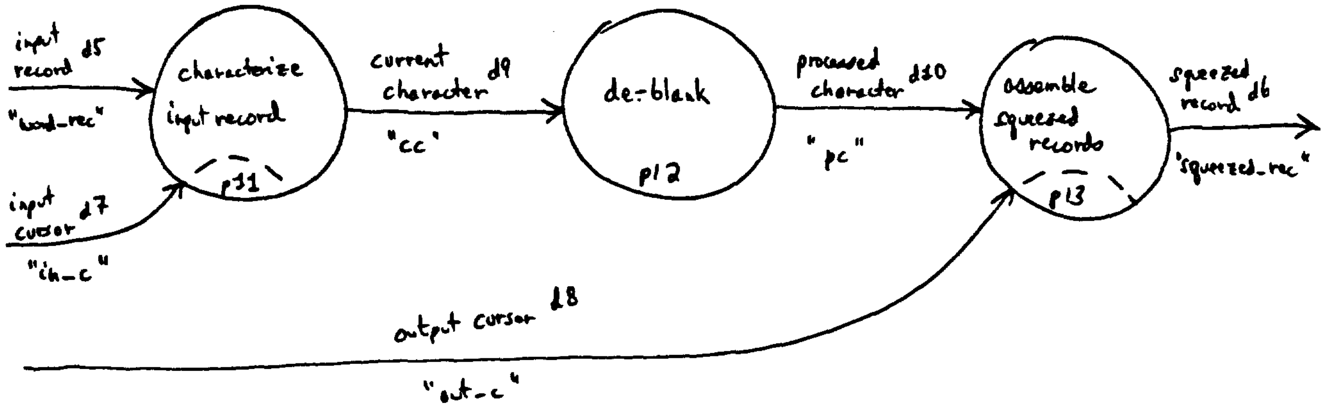

Fig. 8. System specification for p8 (remove excess blanks).

11. In Fig. 9, we see our first example of a two state program. The NEXT commands on Lines 12 and 19 are used to set the state in which the program will "wake up" at its next data-driven activation. A REITERATE command (Line 12) causes immediate execution of the code for a new state during the current activation.

Shown in Fig. 10 is the heart of the squeeze program. The two states are used to remember how many blanks have been seen. This sort of state information can also be stored in a explicit variable, (e.g., d7 and d8 in Fig. 8). However, when the number of states is small, it often seems clearer to use the built-in finite state mechanism. The program specification for p13 (assemble squeezed records), shown in Fig. 11., completes the example system specification.

Note that the since activations of p11, p12, and p13 are data-driven, they can run asynchronously (in parallel). This program is more difficult to design sequentially because of the artificial sequencing imposed by the von Neumann approach to programming.

A more formal discussion of notation and execution semantics is given in [21].

```
 1  /*====================== characterize input record p11 ==*/
 2  /*                                                  [p8]  */
 3  p11(d5,d7,d9)
 4  struct { char word_rec[80] } d5;
 5  struct { int in_c } *d7;
 6  struct { char cc } *d9;
 7  BEGIN(p11)
 8     s0: /* new record or eof */
 9        if(EOF(d5))
10        {  CLEAR(d5); CLEAR(d7); END(d9); }
11        else
12        {  NEXT(s1); REITERATE; }
13     SUSPEND;
14     s1: /* in record */
15        d9->cc = d5.word_rec[d7->in_cc];
16        SET(d9);
17        if( (d7->in_cc++) >= 80)
18        {  /* get new record */
19           CLEAR(d5); CLEAR(d7); NEXT(s0); }
20     SUSPEND;
21  END(p11);
```

Fig. 9. Program specification for p11 (characterize input record).

```
/*========================================== de-blank p12 ==*/
/*                                                   [p8]  */
p12(d9, d10)
struct { char cc } d9;
struct { char pc } *d10;
BEGIN(p12)
    s0: /* no blanks */
        if(EOF(d9))
        {  CLEAR(d9);  END(d10);  }
        else /* character */
        {  d10->pc = d9.cc;  /* copy character */
           if(d9.cc == ' ')
              NEXT(s1);
           CLEAR(d9);  SET(d10);  }
    SUSPEND;
    s1:  /* one blank */
        if(EOF(d9))
        {  NEXT(s0);  REITERATE;  }
        else /* character */
        if(d9.cc == ' ')
        /* eat excess blank */
           CLEAR(d9);
        else
           NEXT(s0);  REITERATE;
    SUSPEND;
END(p7);
```

Fig. 10. Program specification for p12 (de-blank).

```
/*======================== assemble squeezed records p13 ==*/
/*                                                  [p8]  */
p7(d6,d8,d10)
struct { char squeezed_rec[60] } *d6;
struct { int out_c } *d8;
struct { char pc } d10;
BEGIN(p13)
    s0: /* assemble */
        if(EOF(d10))
        /* pad last record */
        {  NEXT(s1); REITERATE }
        else /* copy character to output */
        {  d6->squeezed_rec[d8->out_c] = d10.pc
            if( (d8->out_c++) >= 60)
            {  SET(d6); CLEAR(d8); } }
    SUSPEND;
    s1: /* pad last record */
        if( (d8->out_c) > 0)
        while ( (d8->out_c) < 60)
            d6->squeezed_rec[d8->out_c++] = ' ';
        SET(d6); NEXT(s2);
    SUSPEND;
    s2: /* after last record */
        CLEAR(d8); CLEAR(d10); END(d6); NEXT(s0);
    SUSPEND;
END(p13);
```

Fig. 11. Program specification for p13 (assemble squeezed records).

3. Experience with the Data Activated Programs

A data driven solution is presented in[21] to the Telegram Problem[22] (a more elaborate version of "squeeze") to illustrate the techniques of data driven programming. The solution to the Telegram problem was first implemented in COBOL using a general purpose macro processor (m4) on UNIX. The input to the macro system consisted of macro calls that specify data definitions, **p − d** connections (the system *wirelist*) and program actions. The output of the macro system was standard COBOL code that included system drivers and tables to simulate data driven execution.

The same solution has since been implemented in Pascal and C on two other computer systems. Although the details of each implementation were quite different, each operated identically in a stronger sense than is usually the case for program transportation: the run time state transitions and (abstract) program data flow actions for the three implementations can be shown to be isomorphic.

The techniques have also been applied to the problem of speeding up FORTRAN code execution on vector processor supercomputers, including the CRAY-1 and

CYBER205[23] [24] [25]. The speedups achieved were made possible because of the aid system data flow diagrams give in visualizing and gathering groups of floating point numbers for processing by an "inner loop" **p** that can effectively utilize pipelined floating point arithmetic units. The speedups obtained were an order of magnitude better than those obtainable by use of other techniques, including automatic "vectorizing" programs[26], optimizing compilers, and even hand coded assembler!

4. The Proposed Software Engineering Support Environment

Shown in Fig. 12 is a system diagram of the inter-relationships between the three software tools proposed for this project. The tools are described in more detail below.

4.1. System Designer Interface

An interactive graphics-based system designer interface for display and updating of data flow diagrams, with automatic consistency checking across levels. One of the primary difficulties with this approach to system design is the manual generation and cross-checking of the system data flow diagrams.

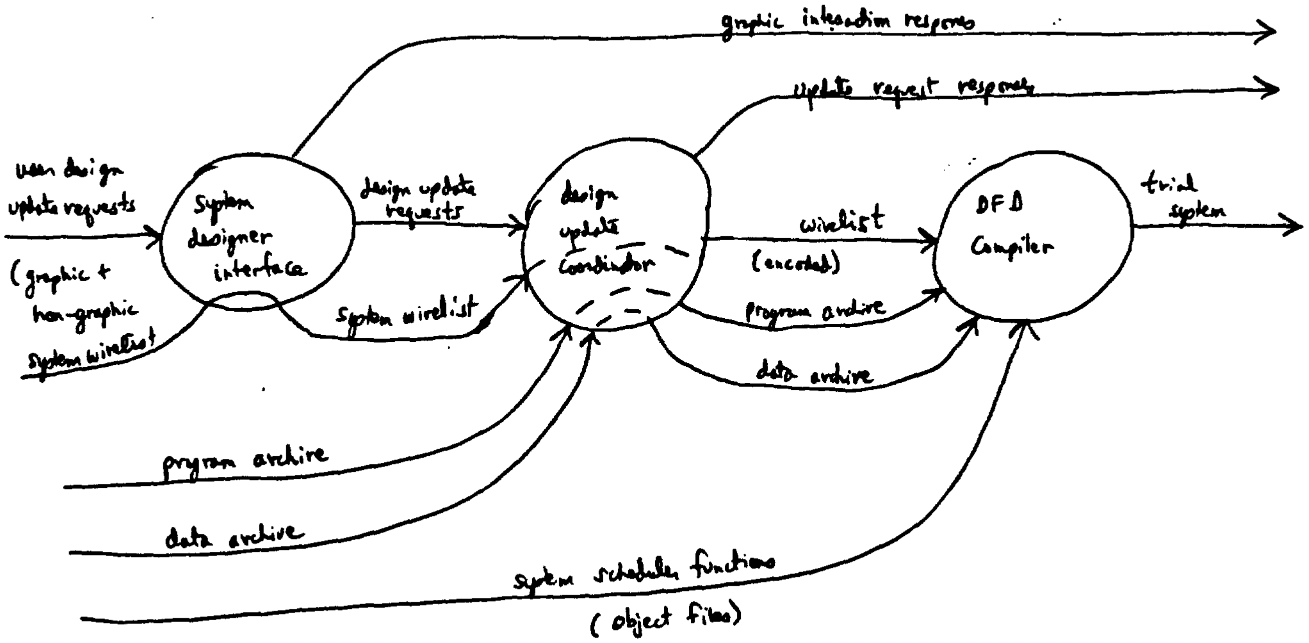
graphic interaction response
update request response
user design update requests
(graphic + non-graphic system wirelist)
System designer interface
design update requests
System wirelist
design update coordinator
wirelist
(encoded)
program archive
data archive
DFD Compiler
trial system
program archive
data archive
system scheduler functions
(object files)

Fig. 12. Proposed software engineering support tools.

4.2. Design Update Coordinator

A tool to manage simultaneous change requests by different designers to an evolving system design. The tool coordinates shared access to a central system design data base, by allowing changes to a **p** only when no associated **d** is "checked out." A similar rule is used to decide whether to allow a person to change a **d**.This tool will use the "advisory lock" facility available on Berkeley UNIX 4.1c to interlock parallel accesses to the the system archive files.

4.3. DFD Compiler

A tool to automatically transform system data flow diagrams into the corresponding data-activated data flow drivers. This task has up until now been performed manually, but is quite tedious and error prone. A preliminary version of this tool exists. The tool can also activate the C compiler and link/loader.

5. Significance of the Proposed Research

A great deal of human effort is now being expended in the production of software, and the amount is increasing. It is becoming apparent that traditional methods for programming are running up against a complexity barrier, and that other models for programming and "systemming" should be investigated. From past experience with prototype data-activated data flow diagram systems, we can make several observations about the potential of this approach. First, the approach is largely compatible with existing coding methods. Up to a certain level, programming can be done in slightly extended versions of familiar programming languages. Second, a significant degree of intellectual leverage seems to be inherent in two dimensional notations for system level design. Third, existing production systems can be transformed "whole" for execution under the data-activated paradigm relatively easily. More detailed data flow modelling of existing code (such as was done for the vector supercomputer experiments) is possible and seems useful, although the difficulty of doing this depends a great deal on how well-written the original code was, and to what level of detail the data flow modelling is carried.

5.1. Objectives

The research proposed here has four long term objectives:

1) *an integrated software engineering facility*— to allow integration of the specification, prototyping, coding, and testing of very large software systems.
2) *requirements/design coherence*— to create an environment for constructing and modifying large-scale systems of programs that allows a model of system requirements to become, via coherent, gradual refinement, an implemented system.

3) *cost-effective re-use of software parts and subassemblies*— to create the software equivalent of plug-in hardware modules: an inventory of reliable, compatible, software templates, and an environment in which they can be easily adapted, instantiated, and interconnected.

4) *data driven parallel processing* — to develop practical methods for the design of reliable systems of cooperating sequential processes[27].

5.2. Related Work

Given below is a summary of ideas related to the current proposal, along with some comments on the relationships:

- *Information hiding*[28] is supported by the proposed system because data paths can be separated into completely independent "universes" linked together by modules which have access only to the information that they need.
- *Data abstractions*[29] can be provided as a natural result of hierarchical refinement in detail of problem solutions.
- *Object-oriented programming* and *message passing systems*[30] somewhat resemble the proposed techniques but are much less restricted. For example, in Smalltalk[31] an object can send a message to any other object at any time. In our model, the possible communication paths are predetermined when a system is designed, and the possible times for communication are restricted at run-time by data flow interactions.
- *Unix Pipes*[32] are an example of a data driven implementation of *coroutines*[33]. The current proposal can be viewed as an extension of the idea of pipes and co-routines to more complex (hierarchical, multiple input/output) networks.
- *Data Flow Diagrams* used by DeMarco[34] and others in Structured Analysis resemble our notation. The primary difference is that traditional data flow diagrams have no direct implementation model. After system requirements have been defined, the data flow diagrams must be transformed into control-oriented designs for implementation[35].
- *Data Flow Programming and Machine Architectures*[36] share many biases about programming with the proposed research. The major difference is that data flow architectures and languages take a very fine grained view of system execution, typically at the level of an arithmetic operator and two operands. We tend to deal with much larger "chunks," corresponding to 5-50 higher-level programming language statements.

6. Development Plan

We intend to develop the tools in the order presented below. While the tools could be developed in parallel (once their exact data interfaces have been defined), we plan to use the DFD compiler tool and the methods proposed herein to help design and build the other two tools. We intend to implement the tools initially in C under UNIX. Past experience with transporting data flow schedulers indicates that it will not be difficult to translate the tools for use on other systems, and/or to change the target implementation language.

The time estimates given reflect our current estimate of the relative difficulty of producing the tools. The DFD Compiler and Archive Coordinator will require only our current OGC VAX/UNIX system for development support. The System Designer Interface will require an interactive graphics system. It is proposed to fund a SUN Workstation with a Bitpad as part of this research for implementation of this tool. The SUN is suitable because it provides a high-performance bit-mapped graphics display, can transfer files from/to the VAX, and runs the same operating system (Berkeley Unix 4.1c).

6.1. DFD Compiler

Development time: 5 months

6.2. Archive Coordinator

Development time: 4 months

6.3. System Designer Interface

Development time: 3 months

7. References

[1] L. A. Belady and M. M. Lehman, *The Characteristics of Large Systems*, IBM Tech. Rep. No. RC 6785 (#28969) Sept. 13, 1977.

[2] M. Fitter and T. R. G. Green, "When do diagrams make good computer languages?" *Int. J. Man—Machine Studies*, vol. 11, 1979, pp. 235-261.

[3] S. B. Sheppard, E. Kruesi, and B. Curtis, "The effects of symbology and spatial arrangement on the comprehension of software specifications," in *Proc. 5th Int. Conf. on Software Engineering*, March 1981, pp. 207-214.

[4] B. Shneiderman, R. Mayer, D. McKay and P. Heller, "Experimental investigations of the utility of detailed flowcharts in programming," *Comm. of the ACM*, vol. 20, no. 6, June 1977, pp. 373-381.

[5] E. W. Dijkstra, "Programming: From craft to scientific discipline", in *Proc. Int. Computing Symposium 1977*, Liege, Belgium (ed. by E. Morlet and D. Ribbens). Amsterdam: North-Holland, April 1977, pp. 23-30.

[6] C. A. R. Hoare, "Software engineering: A keynote address", in *Proc. 3rd Int. Conf. on Software Engineering*, Atlanta, GA, May 1978, pp. 1-4.

[7] R. G. Babb II and L. L. Tripp, "An engineering framework for software standards," in *1980 Proc. Annual Reliability and Maintainability Symposium*, Jan. 1980.

[8] P. Naur and B. Randell (eds.), *Software Engineering*, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, Oct. 1968

[9] J. N. Buxton and B. Randell, *Software Engineering Techniques*, (also sponsored by the NATO Science Committee), Rome, Italy, Oct. 1969.

[10] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming Theory and Practice*. Reading, MA: Addison-Wesley, 1979.

[11] F. DeRemer and H. H. Kron, "Programming-in-the-large versus programming-in-the-small," *IEEE Trans. Software Engineering*, vol. SE-2, no. 2, June 1976, pp. 80-86.

[12] M. A. Jackson, *Principles of Program Design*. London: Academic Press, 1975.

[13] P. Freeman and A. I. Wasserman, *Tutorial: Software Design Techniques*. Los Alamitos, CA: IEEE Computer Society, 3rd. ed., May 1980.

[14] R. G. Babb II, "Coherent realization of system requirements," in *Proc. Int. Symposium on Current Issues of Requirements Engineering Environments*, (ed. by Y. Ohno), Sept. 1982, pp. 103-105.

[15] P. Zave, "An operational approach to requirements specification for embedded systems," *IEEE Trans. on Software Engineering*, vol. SE-8, no. 3, May 1982, pp. 250-269.

[16] J. R. Cameron, "Two pairs of examples in the Jackson approach to system development," in *Proc. 15th Hawaii Int. Conf. on System Sciences*, Jan. 1982, vol. 1., pp. 304-313.

[17] M. V. Zelkowitz and M. Branstad (chairpersons) *ACM SIGSOFT 2nd Software Engineering Symposium: Workshop on Rapid Prototyping*, Columbia, MD, April 1982.

[18] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.

[19] J. Rader (ed.), *Proc. of the First Workshop on the Engineering of VLSI and of Software*, July 1983.

[20] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[21] R. G. Babb II, "Data-driven implementation of data flow diagrams," in *Proc. 6th Int. Conf. on Software Engineering*, Tokyo, Japan, Sept. 1982, pp. 309-318.

[22] P. Henderson and R. Snowdon, "An experiment in structured programming," *BIT*, vol. 12, 1972, pp. 38-53.

[23] R. G. Babb II and J. M. Hardy, "Applications of Program/System design techniques to the CRAY-1," paper presented at the 1980 ACM Mountain Region Conference, Denver, Nov. 15, 1980.

[24] J. M. Hardy and R. G. Babb II, "Speedup of FORTRAN subprograms ROTATE and ALAG3D on the CRAY-1 using Program/System methods," Final Report for Optical Systems Branch, Air Force Weapons Laboratory, Contract F2965079, D0028-5019, Summers Engineering, Inc., October 1980.

[25] J. M. Hardy and R. G. Babb II, "A Program/System solution to the EOS (Equation of State) Problem," Final Report for Lawrence Livermore National Laboratory under University P.O. 3196401, Summers Engineering, Inc., May 1981.

[26] D. J. Kuck, R. H. Kuhn, B. Leasure and M. Wolfe, "The structure of an advanced vectorizer for pipelined processors," in *Proc. COMPSAC-80*, Oct. 1980, pp. 709-715.

[27] C. A. R. Hoare, "Communicating Sequential Processes," *Comm. of the ACM*, vol. 21, no. 8, Aug. 1978, pp. 666-677.

[28] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Comm. of the ACM*, vol. 15, no. 12, Dec. 1972, pp. 1053-1058.

[29] B. Liskov and S. Zilles, "An introduction to formal specifications of data abstractions," in *Current Trends in Programming Methodology*, vol. 1: *Software Specification and Design.* Englewood Cliffs, NJ: Prentice-Hall, 1977.

[30] T. Rentsch, "Object Oriented Programming," *SIGPLAN Notices*, vol. 17, no. 9, Sept. 1982, pp. 51-57.

[31] Special Issue of *BYTE* on Smalltalk. Aug. 1981.

[32] B. W. Kernighan and J. R. Mashey, "The UNIX^{TM} programming environment," *Software—Practice and Experience*, vol. 9., no. 1, Jan. 1979, pp. 1-15.

[33] M. E. Conway, "Design of a separable transition-diagram compiler," *Comm. of the ACM*, vol. 6, no. 7, pp. 396-408, July 1963.

[34] T. DeMarco, *Structured Analysis and System Specification.* New York, NY: Yourdon, 1978.

[35] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* (2nd ed.) New York, NY: Yourdon Press, 1978.

[36] T. Agerwala and Arvind (eds.), Special Issue of *Computer* on Data Flow Systems, vol. 15, no. 2, Feb. 1982.