Hardware and Software for Real Time Process Control
J. Zalewski and W. Ehrenberger (Editors)
Elsevier Science Publishers B.V. (North-Holland)

Automatic Documentation of Programs

Jürgen F.H. Winkler

Siemens AG, Corporate Research, Munich, FRG

Industrial software products, including those for real-time applications, are typically families of large, complex, and long living programs. They are therefore quite costly and are usually for most of their lifetime in the maintenance phase. Maintenance depends heavily on documentation that is accurate and up-to-date. Some aspects of programs, that are important for maintenance, cannot be expressed directly in contemporary programming languages.

This paper deals with the aspects of automatically generated program documentation and points out which parts of the program documentation can be generated automatically. It reports on a tool, PasDok-X2, that implements these ideas for an extended form of Pascal. PasDok-X2 generates the following items: complete interface descriptions, structure charts, and hints to possible program deficiencies.

Keywords: program maintenance, program documentation, automatic documentation, program analysis, modular Pascal

1 Introduction

Programs embedded in industrial products are typically rather large but must still be of high quality. This holds especially for programs in real-time systems because often life and welfare of single persons or society as a whole depend on such systems. Examples for such systems are: systems for air traffic control, for control of equipment in the intensive care unit of a hospital, or for control of anti-skid brakes in an automobile. For more examples see Peter G. Neumanns regular column "Risks to the public in computers and related systems" [SEN 12,4(1987)2..18; 12,3(1987)2..17; 12,2(1987)4..22; 12,1(1987)3..28].

The development costs of such high quality programs are quite high, and therefore it is necessary that they are used for quite a long time in order to be economically feasible. Due to this well known reason such programs are during almost all of their lifetime in the maintenance phase. The maintenance task is in the worst, but not totally unusual, case characterized by:

- the person that has to perform a program modification does not know the program at all.
- there exists no documentation apart from the program itself. Often some documentation exists that is not completely up to date.

I.e. the task involves the understanding of a completely unknown program.

What is the best support for this understanding of an unknown program? The source text of the program is necessary [Wei 87] but not sufficient because it does not facilitate the understanding of the coarse structure of the program and because it contains important relationships, as e.g. that between definition and use of an entity,

only implicitely. Therefore, additional documentation is necessary, that contains such information as mentioned above. This additional information must be kept up-to-date during the maintenance phase. This is easy if this information is produced automatically, and tedious and error prone otherwise.

In the work reported in this paper we focus on those phases of the SW life cycle in which programs are developed and manipulated. The other phases of SW development are also important but in case of doubt the program is the crucial document because it determines the behaviour of the computer [Wei 87]. Techniques that model the whole development process of a program in a program derivation [SS 83] will have a long way to go until they are practically useful. On the other hand, the approach and techniques presented in this paper can also be applied to other formal descriptions that are used during SW development [Jac 86].

The paper is organized as follows: Section 2 deals with program evolution and the need to understand unknown programs. The deficiencies of contemporary programming languages with respect to documentation are the topic of section 3. Section 4 presents our approach to automatic program documentation by source code enrichment, and a tool, PasDok-X2, that implements this approach for modular Pascal programs is presented in section 5. Section 6 summarizes the paper and offers some conclusions and topics of further work.

2 Program Evolution: The need to understand unknown programs

As already mentioned in the introduction, industrial program products are characterized by the following properties:

- they are quite large (10^5 - 10^7 loc)
- they have a rather long lifetime (10 - 30 years)
- they must be of high quality
- they are produced by multi-person teams
- they are often produced in multi-site projects
- they are during almost all of their lifetime in maintenance

One important consequence of these characteristics is that such large, long-lived programs are usually produced in an evolutionary approach. After the first version, that is often built from scratch, they are gradually improved and extended. During their lifetime they evolve in a number of successive versions and alternative variants [Win 87, 88]. Typical lifetimes of such programs comprise 10 - 30 years. Due to such long lifetimes and other factors, as e.g. turnover of personel and organzational separation of development and maintenance, it will often occur that a person has the task to modify a program that s/he has never seen before.

A key issue in program evolution is therefore the need to understand unknown programs. Several factors influence program understanding:

- knowledge and experience of the SW engineer (SWE)
- programming language used
- programming style
- quality of documentation
- tool support for analyzing and scrutinizing the program

This tool support should provide the SWE with the following information:

① *Overviews of the program.* Such overviews give e.g. the static structure of the program in varying detail. For an Ada® program each compilation unit can be depicted as a node in a graph whose edges represent the different relationships existing between compilation units [Win 82]. For monolithic Pascal programs an overview is the nesting structure [LSW 87; Luc 87]. Another overview is the call graph that shows all possible calls of subprograms as a relation over the set of subprograms [LSW 87].

② *Dataflow in the program.* One form is a relation over the sets of expressions in the program that indicates which expressions depend on which other expressions. Another form is the program slice [Wei 82], that collects all those parts of a program that influence the value of a certain variable.

③ *Complete interface descriptions.* In most programming languages a nested subprogram or block can use global entities without stating this fact explicitely. This can make the understanding of the program pretty hard. Therefore the subprogram heading should be completed by a documentation of such global uses [LSW 87].

④ *Hints to flaws in the program.* Examples for such flaws are the occurrence of literal constants outside of constant declarations or are entities that are declared but never used.

⑤ *Def-use-relation.* E.g. cross reference list.

This additional information can be presented to the SWE in two different ways:

① As additions to the program source text. In this case we speak of automatic documentation of the program because this additional information is essentially a form of documentation. This documentation is quite close to the program source text.

② On demand on the screen of a terminal. In this case the program is conceived as a highly structured object (similarly as in the hypertext approach [Con 87]) that is too complicated to be presented in all aspects by one view on a planar medium. The tool therefore provides a number of different views on the program corresponding to the different aspects (properties) of the program.

3 Deficiencies of Programming Languages

Most of the additional information mentioned in sect. 2 cannot be expressed in contemporary programming languages. We make a difference here between the noncomment parts of a program and the comment parts. In the following we speak of the noncomment part as of the program in order to clearly distinguish between comment and noncomment. Both together are called the source text. Clearly, the additional information mentioned above can be represented in comments. But comments are not checked for consistency and correctnes by the language processor, and therefore, the information contained in comments is much less trustworthy than that in the program. The programmer may e.g. give a complete interface description of a procedure P by adding the global entities used in P in a comment:

```
PROCEDURE P (Par1: T1;  Par2: T2);
                    { Globals used: G1, G2, G3 }
```

This comment may be correct when first written by the person that writes also the body of P. But during maintenance the body of P may be changed such that P does now also refer to the global entitiy G4. If the maintenance programmer is not aware of the comment in the procedure heading or simply forgets to bring the comment up-to-date then program and comment are no longer consistent. Since comments are skipped by the language processor this inconsistency may exist for an arbitray long period of time.

Similar arguments hold for other kinds of information mentioned in sect. 2. Overviews, structural relations, and dataflow information can only be inserted in the source text as comments. On the other hand this information is already contained implicitely in the program. Therefore, we need additional tools to derive this information from the program and present it in a form that can be conveniently used by the SWE.

4 Automatic Documentation by Program Enrichment

We speak of automatic documentation if some useful information is derived mechanically from the program and included in the source text. If the information is presented to the SWE in an interactive way we speak of a program information system.

In automatic documentation we distinguish between two approaches:

① The information is added to the source text such that a new document is created that cannot be processed by the language processor (e.g. compiler).

② The information is inserted in the source text such that the new source text can still be processed by the language processor. Usually this can be done by putting the additional information as comments into the source text. In the rest of the paper this technique is called program enrichment. In the following we designate the documented form of a program P by P_{doc}.

Program enrichment has the advantage that P_{doc} can still be used as the basis of further development (maintenance). The SWE deals with one document only, and this fact facilitates his work.

In order to be effective, program enrichment must adhere to some principles. The most important of them are:

- *Recognizability*: the SWE must recognize his original program P easily in P_{doc}. We therefore do not perform prettyprinting as part of the documentation process.
- *Adequate placement:* the information that is added to P must be put at those places where the SWE expects it. The list of global uses e.g. is best inserted immediately after the subprogram heading, because this list and the subprogram heading together form the interface between the subprogram and its environment.
- *No pollution of the source text*: it is important that the original program is not buried under a mass of additional information because the SWE will then have difficulties to recognize his program. In PasDok-X2 (see sect. 5) P_{doc} is about 15 - 20% larger than P.
- *Ergonomic layout of* P_{doc}: the additional information must be inserted in such a way into P that it is easily spotted. In PasDok-X2 we have put those parts of the documentation that annotate single lines of the source text at the right margin in order that they stand out and can be easily recognized. As an example we take the following annotation:

```
Write('End of input: <S>');
ReadLn(Ch);
X := 13.25;{% !!! Literal }
IF Ch <> 'S' THEN BEGIN
                    Einlesen( .... );
```

In this form the annotation {% !!! Literal}, which indicates that there occurs an assignment to a global variable (!!!) and a literal constant (Literal), cannot be spotted at a first glance. If the strategy of PasDok-X2 is applied we obtain:

```
Write('End of input: <S>');
ReadLn(Ch);
X := 13.25;                          {% !!! Literal}
IF Ch <> 'S' THEN BEGIN
                    Einlesen( .... );
```

In this form the SWE finds the hints generated by the documentation tool by scanning the right marging of P_{doc}.

5 PasDok-X2

PasDok-X2 is a tool that implements the approach to automatic documentation of programs as it was developed in the preceding sections. PasDok-X2 documents

Pascal-XT programs. Pascal-XT [Sie 86] is an extension of ISO-Pascal that contains a module concept similar to Ada and Modula-2.

PasDok-X2 accepts correct Pascal-XT programs and generates two different program documents whose detailed structure can be controlled by numerous commands and options:

(1) **The file version** (enhanced source text)

This document is used in program development and can always be compiled since it is a correct Pascal program. This requirement forces PasDok-X2 to insert the additional information into the source text as normal program comments.

At the beginning of a program unit the static nesting is displayed, and at the end the call graph. After the heading of each subroutine, a complete interface documentation is inserted. In cases where the type of a variable declaration is not simple, the line number of the corresponding type declaration is given. Program deficiencies as e.g. side effects, occurrences of literals, and identifiers, that are never used, are also flagged.

The file version is generated for a single compilation unit. We hope that this type of enhanced source will turn out to be useful for initial program development, and especially for program maintenance.

(2) **The paper or book version**

The paper or book version generated by PasDok-X2 is composed of a table of contents, of the enhanced source text (file version) and of further parts (appendices). These contain a summary of the program modules and of their relations, especially the "uses" relation, and a short summary for each compilation unit. A further optional appendix contains a table of all program deficiencies detected in the program. For each procedure a summary of its important formal properties may be generated. Furthermore, statistic tables as well as a cross reference list may be printed out.

The book version can also be generated for a complete Pascal-XT program consisting of several compilation units. All parts or entities are referenced by page and line numbers. This document, which is not anymore a syntactically correct Pascal program, should be the basis of those parts of the program documentation, which have to be prepared by hand because they cannot be generated automatically.

Fig. 1 shows the relations between PasDok-X2 and the different program versions.

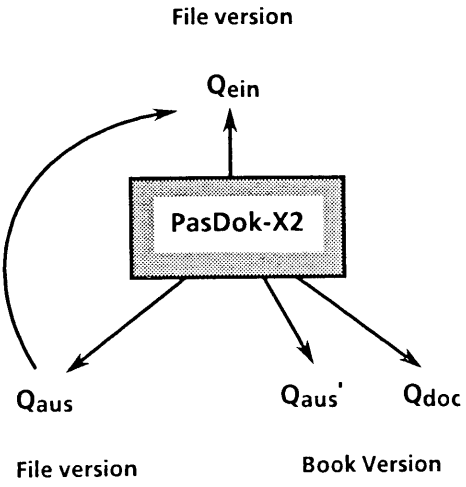

Fig. 1. PasDok-X2

Example for the use of PasDok-X2

```
PACKAGE KELLER(OUTPUT);
TYPE ELEMENT =
      RECORD NAME : ARRAY[1..20] OF CHAR;
             NUMMER : INTEGER;
      END;
FUNCTION K_ISTLEER : BOOLEAN;
PROCEDURE PUSH(ELEM : ELEMENT);
PROCEDURE POP(VAR ELEM : ELEMENT);

END.
```

package KELLER

```
with keller; from KELLER use element;
package menge(output);
function m_istleer : boolean;
procedure istelement(elem : element;
                     var b : boolean;
                     var platz : integer);
procedure einfuegen(elem : element);
procedure streichen(elem : element);
var mzeiger : 0..50;
end.
```

package MENGE

```
WITH MENGE,KELLER;
FROM KELLER USE ELEMENT, PUSH, POP;
FROM MENGE  USE EINFUEGEN, STREICHEN;

PROGRAM TRANSFER ( INPUT,OUTPUT,OUTPT);
(*! DIES IST EIN BEISPIEL MIT EINEM
 *  HAUPTPROGRAMM UND ZWEI PAKETEN     *)

LABEL 9999 ,
      123 ;
CONST MAX1 = 1000; hmax = 0;
      MAX2 = 1234;
TYPE FELD = ARRAY [1..MAX1] OF 1..10;
     VERB = RECORD SCHLUESSEL : INTEGER;
                   FF : FELD;
                   PEGEL : 1..MAX1;
            END;
VAR OUTPT : FILE OF VERB;
    A :FELD;
    ELLI : ELEMENT;
    I, J : INTEGER;
    X    : REAL;

PROCEDURE EINGABE;
(*! EINGABE BELIEBIG VIELER ELEMENTE *)
VAR CH : CHAR;
    EL : ELEMENT;

PROCEDURE EINLESEN (VAR ELEM:ELEMENT);
 TYPE DUMMY = (ROT,GELB,GRUEN);
 VAR I : INTEGER;
     ZK : STRING;
     KORREKT : BOOLEAN;
 BEGIN
   KORREKT := FALSE;
   REPEAT
    WRITE('NAME :');READLN(ZK);
    WRITE('ZAHL :');READLN(I);
    KORREKT := (I>0)AND(I<=MAX1);
   UNTIL KORREKT;
   ELEM.NAME := ZK; ELEM.NUMMER := I;
 END; (*EINLESEN*)
BEGIN
  REPEAT
   WRITE('Ende der Eingabe : <S> ');
   READLN(CH);
   X := 13.25;
   IF CH<>'S' THEN BEGIN EINLESEN(EL);
                         PUSH(EL)
                   END;
  UNTIL CH = 'S';
END;

PROCEDURE UEBERTRAG;
VAR ELEM : ELEMENT ;
BEGIN
  WHILE NOT KELLER.K_ISTLEER DO
     BEGIN
       POP(ELEM);
       EINFUEGEN(ELEM);
     END;
END;
BEGIN (* HAUPTPROGRAMM *)
  I := I-1;
  EINGABE;
  UEBERTRAG;GOTO 123;
123: WRITELN(MENGE.MZEIGER,' ELEMENTE');
END.
```

program TRANSFER

```
{% Statische Schachtelung :
DEF    NAME
-------------------------------------------------------
   16  TRANSFER(INPUT,OUTPUT,OUTPT);
   35    EINGABE;
   46      EINLESEN(VAR ELEM : ELEMENT);
   72    UEBERTRAG;
-------------------------------------------------------}
WITH KELLER, MENGE;
FROM KELLER USE PUSH, POP, K_ISTLEER, ELEMENT;
FROM MENGE  USE EINFUEGEN, STREICHEN;

PROGRAM TRANSFER ( INPUT,OUTPUT,OUTPT);
(*! DIES IST EIN BEISPIEL MIT EINEM
 *  HAUPTPROGRAMM UND ZWEI PAKETEN   *)

LABEL 9999 ;                              {%nicht benutzt}
      123 ;                                    {%==> 0090}
CONST MAX1 = 1000;
      MAX2 = 1234;                        {%nicht benutzt}
TYPE FELD = ARRAY [1..MAX1] OF 1..10;          {%Literal}
     VERB = RECORD SCHLUESSEL : INTEGER;
                   FF : FELD;
                   PEGEL : 1..MAX1;          {%Literal}
            END;
VAR OUTPT : FILE OF VERB;                        {%-->25}
    A :FELD;                                     {%-->24}
    ELLI : ELEMENT;                                {%III}
    I,J  : INTEGER;                         {%nicht benutzt}
    X : REAL;

{%#################################################}
PROCEDURE EINGABE;
{%=================================================
   | Schnittstelle von EINGABE :
   | globale   : ELEMENT(T), X(V),
   | Proz/Funk : PUSH(P),
   =================================================}
(*!  EINGABE BELIEBIG VIELER ELEMENTE *)
VAR CH : CHAR;
    EL : ELEMENT;                                  {%III}

{%#################################################}
  PROCEDURE EINLESEN (VAR ELEM:ELEMENT);           {%III}

   TYPE DUMMY=(ROT,GELB,GRUEN);{%Literal/nicht benutzt}
   VAR I : INTEGER;
       ZK : STRING;
       KORREKT : BOOLEAN;
   BEGIN                                        {% einlesen}
     KORREKT := FALSE;
     REPEAT WRITE('BITTE NAME EINGEBEN :');READLN(ZK);
            WRITE('BITTE ZAHL EINGEBEN :');READLN(I);
            KORREKT := (I>0)AND(I<=MAX1);       {%Literal}
     UNTIL KORREKT;
     ELEM.NAME := ZK; ELEM.NUMMER := I;
   END;                                         {% einlesen}
BEGIN                                            {% eingabe}
  REPEAT
    WRITE(' ENDE DER EINGABE : <S>');
    READLN(CH);
    X := 13.25;                                {%!!! Literal}
    IF CH<>'S' THEN BEGIN EINLESEN(EL);          {%Literal}
                          PUSH(EL)
                    END;
  UNTIL CH = 'S';                                {%Literal}
END;                                             {% eingabe}
```

File version of TRANSFER (Q_{out}) (cont'd)

```
{%#################################################}
PROCEDURE UEBERTRAG;
{%=================================================
   | Schnittstelle von UEBERTRAG :
   | glob. Gr. : ELEMENT(T),
   | glob. UP. : EINFUEGEN(P), K_ISTLEER(F), POP(P),
   =================================================}
VAR ELEM : ELEMENT ;                              {%III}
BEGIN                                          {%uebertrag}
  WHILE NOT K_ISTLEER DO                          {%III}
      BEGIN
        POP(ELEM);
        EINFUEGEN(ELEM);
      END;
END;                                           {% uebertrag}
BEGIN                                           {% transfer}
  J := I-1;                          {%Literal/nicht init.?}
  EINGABE;
  UEBERTRAG;GOTO 123;                             {%==>0090}
123: WRITELN(MENGE.MZEIGER,' ELEMENTE');
END.                                           {% transfer}
{%
AUFRUFGRAPH :

NR   RUFER    NAME
-------------------------------------------------------
1       -     TRANSFER
2       1       EINGABE
3       2         EINLESEN
4       2         PUSH
5       1       UEBERTRAG
6       5         K_ISTLEER
7       5         POP
8       5         EINFUEGEN
-------------------------------------------------------}
```

file version of TRANSFER (Q_{out})

The example presented on the previous pages is a program consisting of three compilation units:

- the main program TRANSFER (p.13), and
- two packages KELLER and MENGE (p.12), of which only the specification parts are shown.

MENGE refers to KELLER, and TRANSFER refers to both MENGE and KELLER. The documented version Q_{out} of TRANSFER is depicted on pages 14 and 15. The annotations occur as comments at the right margin. At the beginning of Q_{out} we see the static nesting, and at the end the call graph.

6 Summary and Outlook

The relatively short history of programming and programming languages is characterized by the initial development of new programs [DDH 72; Wir 71]. Research and education in computer science has focused mainly on the initial development of rather small programs [Dij 76].

The situation in industry is somewhat different, and is characterized by the fact that program products are often families of large, modularized, and long living programs [Win 87]. For these program families in industry maintenance is much more important than the initial development [Pro 85].

Apart from compiler and editor a documentation tool as presented in this paper is one of the most important tools for the SWE. PasDok-X2 constructs automatically a documented version of any Pascal-XT program, where this program may also consist of several compilation units.

This documentation tool is very useful because the information added to the program source text is mechanically derived from the program, and therefore, the documented version of a program is always up to date. This is important for large programs, because the additional information is especially useful for such programs, and because the economic advantages of automatic documentation are quite big for large programs.

On the other hand, the information computed by the documentation tool contains also data that is useful for the development and optimization of compilers. Examples are the usage of global entities and nesting levels.

The techniques of automatic documentation, embodied in PasDok-X2, can also be applied to other programming languages as e.g. Ada, Chill, and Modula-2.

With respect to the trend to replace paper and pencil by screen and keyboard it is important to develop more interactive versions of such documentation tools, that assist the maintenance programmer

- in understanding and finding his way through a mostly unknown program,
- and in modifying a mostly unknown program.

Such an interactive tool can rather be called a program information system. It would not only produce complete documents but also answer interactively questions about formal properties of the program the programmer is working on.

Acknowledgments

The work reported here was done in cooperation with the University of Karlsruhe. PasDok-X2 was developed by J Lucas as a masters thesis project which was supervised by A Schmitt and the author. The example in sect. 5 is from [LSW 87].

7 References

Con 87 Conklin, Jeff: A Survey of Hypertext. MCC Techn. Report No. STP-356-86, Rev.2, MCC, Austin, Texas, Dec. 1987.

DDH 72 Dahl, O.-J.; Dijkstra, E.W.; Hoare, C.A.R.: Structured Programming. Academic Press, London etc. 1972.

Dij 76 Dijkstra, Edsger W.: A Discipline of Programming. Prentice Hall, Inc. Englewood Cliffs, N.J., 1976.

Jac 86 Jackson, M.A.: Keynote Address for IFIP Congress 86. In: Kugler, H.-J. (ed): Information Processing 86. North Holland, Amsterdam etc. 1986, pp. xxix..xxxvii.

LSW 87 Lucas, J.; Schmitt, A.; Winkler, J.F.H.: Ein Werkzeug für die Wartung und automatische Dokumentation modularer Pascal-Programme. In: Scheibl, H.-J. (Hrsg.): Software-Entwicklungs-Systeme und -Werkzeuge. Techn. Akademie Esslingen, 1987.

Luc 87 Lucas, Jürgen: Software-Werkzeuge für die Wartung und automatische Dokumentation von Pascal-XT-Programmen. Diplomarbeit, Univ. Karlsruhe, 1987.

Pro 85 Proceedings of the Conference on Software Maintenance 1985. IEEE Computer Societey Press, Washington, 1985.

SEN Software Engineering Notes. Newsletter of ACM SIGSoft.

Sie 86 Siemens AG: Pascal-XT V1.0A (BS2000) Beschreibung und Benutzerhandbuch. München, 1986.

SS 83 Scherlis, W.L.; Scott, D.S.: First Steps towards inferential programming. In: Mason, R.E.A. (ed.): Information Processing '83. Elsevier Science Publishers B.V (North Holland), Amsterdam, 1983, pp. 199..212.

Wei 82 Weiser, Mark: Programmers Use Slices When Debugging. CACM 25,7 (1982) 446..452.

Wei 87 Weiser, Mark: Source Code. Computer 20,11(1987)66..73.

Win 82 Winkler, J.F.H.: Ada: die neuen Konzepte. Elektron. Rechenanlagen 24,4(1982) 175..186.

Win 87 Winkler, J.F.H.: Configuration Control in Families of Large Programs. In: Proc. of the 9th Int. Conf. on SW Engineering, Monterey, 1987, pp.150..161.

Wir 71 Wirth, Niklaus: Program Development by stepwise Refinement. CACM 14,4 (1971) 221..227.

® Ada is a registered trademark of the US Government (Ada Joint Program Office).

Jürgen F.H. Winkler
Siemens AG
Corporate Research and Technology
Otto-Hahn-Ring 6
D-8000 Munich 83
Fed. Rep. of Germany
winkler@ztivax.siemens.com

HARDWARE AND SOFTWARE FOR REAL TIME PROCESS CONTROL

Proceedings of the IFIP WG 5.4/IFAC/EWICS Working Conference on Hardware and Software for Real Time Process Control
Warsaw, Poland, 30 May – 1 June, 1988

edited by

Janusz ZALEWSKI
Institute of Atomic Energy
Otwock/Swierk, Poland

Wolfgang EHRENBERGER
Gesellschaft für Reaktorsicherheit
Garching, Fed. Rep. Germany

1989

NORTH-HOLLAND
AMSTERDAM • NEW YORK • OXFORD • TOKYO

ISBN: 0 444 87127 6

Published by:

ELSEVIER SCIENCE PUBLISHERS B.V.
P.O. Box 103
1000 AC Amsterdam
The Netherlands

Sole distributors for the U.S.A. and Canada:

ELSEVIER SCIENCE PUBLISHING COMPANY, INC.
52 Vanderbilt Avenue
New York, N.Y. 10017
U.S.A.

Library of Congress Cataloging-in-Publication Data

IFIP WG 5.4/IFAC/EWICS Working Conference on Hardware and Software for Real Time Process Control (1988 : Warsaw, Poland)
Hardware and software for real time process control.

Bibliography: p.
Includes index.
1. Process control--Data processing--Congresses. 2. Real-time data processing--Congresses. I. Zalewski, Janusz. II. Ehrenberger, W. D. III. International Purdue Workshop on Industrial Computer Systems. IV. International Federation of Automatic Control. V. European Workshop on Industrial Computer Systems. VI. Title.
TS156.8.I35 1988 670.42'7 88-30910
ISBN 0-444-87127-6

PRINTED IN THE NETHERLANDS