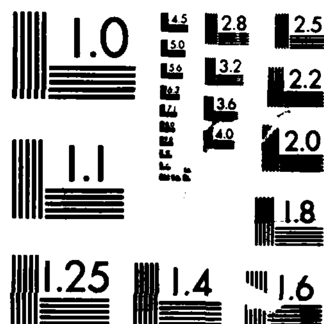
AD-A187 450   AN APPROACH TO THE HIERARCHICAL SPECIFICATION DESIGN AND VERIFICATION OF (U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) B D BRAMSON JUL 87   1/1

UNCLASSIFIED   RSRE-MEMO-4063 DRIC-BR-103381   F/G 12/5   NL

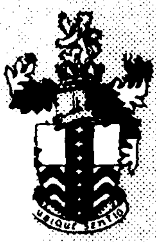


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

DTIC FILE COPY

(2)

UNLIMITED



AD-A187 450

RSRE MEMORANDUM No. 4063

ROYAL SIGNALS & RADAR ESTABLISHMENT

AN APPROACH TO THE HIERARCHICAL SPECIFICATION, DESIGN AND VERIFICATION OF SOFTWARE

Author: B D Bramson

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
R S R E MALVERN,
WORCS.

DTIC
ELECTE
NOV 0 4 1987
S D
E

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4063

Title: AN APPROACH TO THE HIERARCHICAL SPECIFICATION, DESIGN AND VERIFICATION OF SOFTWARE

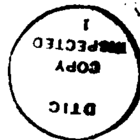
Author: B D Bramson

Date: July 1987

SUMMARY

A method is proposed for the production of large software systems via hierarchies of descriptive levels. The method is based upon the simultaneous refinement of data and code and guarantees conformity between different levels. It is illustrated using the MALPAS Intermediate Language.

Accession For
NTIS GRA&I ☒
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1



Copyright
C
Controller HMSO London
1987

CONTENTS

1 Introduction
2 Data refinement
3 Starting the drive-by-wire car
4 Concluding remarks
References
Appendix: the complete example

1 INTRODUCTION

A recent report [1] described tools for the specification, design, analysis and verification of software. At the same time a technique was suggested for producing large software systems using a hierarchical development method. The purpose of this memorandum is to discuss that method in greater detail and to provide a "toy" example.

The method itself features:

a top down approach,
a hierarchy of descriptive levels,
rapid prototyping,
early customer visibility,
and a verified design.

Support for this is required on at least two fronts. The first is a language, strong in both declarative and algorithmic power, for expressing the specifications and designs of systems in such a way as to allow for the refinement of both data and code. The second is an automatic tool, typically a compliance analyser or a verification condition generator, for comparing programs with their specifications.

The concept of developing software using a hierarchical method has been widely canvassed [3,4,5] but there are few practical systems that offer automatic support. The benefits of early and rapid prototyping are well known [2]. In particular, catching errors during the early stages of design is far cheaper than suffering from them later. (Orders of magnitude have been quoted.)

To illustrate the approach I have employed the input language MALPAS IL from the Malvern Program Analysis Suite. (The fruits of early debates appear in [6].) In particular, IL features abstract types, records and refined types, abstract functions and user-defined rewrite rules, and procedures whose specifications may be used before their bodies are implemented. Further, the MALPAS Compliance Analyser has been employed to check implementations against specifications. For a loop-free procedure this reveals the overlap between the pre-condition and the negation of the weakest pre-condition. In the presence of loops, corresponding information is supplied for each loop-free part. The analyser also checks the pre-conditions of any called procedure. In short, it finds dangerous inputs. (The weakest pre-condition, defined to be the domain of proc ==> post, comprises those inputs for which the post-condition is satisfied.)

The strategy is to use IL for specification and design; to work at high levels of abstraction for as long as possible; and to proceed to lower levels only by agreement with the customer. The method presented here features a pair of hierarchies, the one declarative and the other algorithmic. The declarative part comprises: a hierarchy of types with data refinement via records; a hierarchy of functions with function refinement via replacement rules; and a hierarchy of procedures specified by functions. The algorithmic part comprises a corresponding hierarchy of procedure bodies, refinement being achieved via calls to lower levels.

For a simple system with n (>= 2) levels of description, the production sequence takes the form:

```
BEGIN                                                         (1.1)
   declare level(1) types and functions;
   specify level(1) procedures via level(1) types and functions;
   enter trivial level(1) bodies;
   perform verification and customer review;

   FOR i FROM 2 TO n
   DO BEGIN
         declare level(i) types and functions;
         refine level(i-1) types and functions;
         specify level(i) procedures via level(i) types and functions;
         design level(i-1) bodies to replace trivial bodies;
         enter trivial level(i) bodies;
         perform verification and customer review
       END
END
```

At level(1), the system is modelled via abstract types and functions, declared without elaboration and used to specify the procedures at that level. Trivial procedure bodies are supplied by equating them to their specifications.

At level(i) (i >= 2) further types are declared to support the refinement of the level(i-1) types via the language constructs REFINED TYPE and RECORD. Further functions are also declared and these play two roles. First, they enable semantic meanings to be assigned to the level(i-1) functions via user-defined rewrite rules and specifically the construct REPLACE. Secondly, along with the level(i) types, they specify the procedures at that level.

The rewriting of the bodies of the level(i-1) procedures involves three processes. First, OUT parameters are decomposed into their constituent fields. Secondly, calls are made to level(i) procedures to modify those fields. Thirdly, the parameters are reconstructed.

After the ith iteration (i >= 1) the result is an IL text, capable of both customer inspection and automatic verification. Bodies of procedures are defined in terms of calls to procedures at lower levels in the hierarchy, except for those at level(i) whose bodies are equated trivially to their specifications. The process terminates as soon as sufficient refinement has taken place for a translation to be made from the lowest level of design to the chosen programming language. (Neither this activity, nor that of translating back into IL prior to verifying the implementation, will be pursued here.)

2 DATA REFINEMENT

Since the method to be presented leans heavily on the technique of data refinement, I present a brief description of that process within the context of MALPAS IL. A simple, geometric example will suffice.

Suppose we require a package to perform transformations on points in a plane. We might start by declaring

```
        TYPE point;                                           (2.1)
```

and some abstract functions for rotating and reflecting points. Later, as the design evolves, we may need to specify the meaning of point in

more detail. Typically, we might substitute

```
TYPE point = RECORD x, y: real ENDRECORD;                    (2.2)
```

for the declaration (2.1). Implicitly, this declares a pair of projection operators X and Y, delivering the components of point, together with a construction function, make_point, that re-assembles the pieces. Specifically, the syntax is defined by:

```
PREFIX X(point): integer;
PREFIX Y(point): integer;
FUNCTION make_point(integer, integer): point;
```

while the following rules define the semantics:

```
REPLACE (x, y: integer)
   X make_point(x, y)
BY x;

REPLACE (x, y: integer)
   Y make_point(x, y)
BY y;

REPLACE (p: point)
   make_point(X p, Y p)
BY p;
```

However, the declaration

```
REFINED TYPE point = RECORD x, y: real ENDRECORD;            (2.3)
```

following that of (2.1) obviates the need for replacing (2.1) by (2.2). For declarations between (2.1) and (2.3), the components of point are inaccessible. Following (2.3) they become accessible. (Of course, point must not be declared again.)

3 STARTING THE DRIVE-BY-WIRE CAR

Suppose that the chief designer of a software controlled motor vehicle has made a request to the software engineers that:

starting the car is to be intercepted
by a safety-unit (the guard) (3.1)
and inhibited if unacceptable data is received.

This hypothetical example serves to illustrate the salient points of the hierarchical specification and design method. The car should not start unless it is safe to do so. A refinement of the informal requirement (3.1) follows shortly. Even so, it already contains sufficient information for constructing the top-level IL specifications:

```
[LEVEL(1) SPECS]

TITLE pre_start_checks;

TYPE guard, data;                                            (3.2)
FUNCTION raised(guard): boolean;
FUNCTION lower(guard, data): guard;
```

The level(1) description declares: types "guard" and "data"; a function "raised" to specify that a guard shall fully inhibit ignition before starting the car is even attempted; and a function "lower" to change the state of a guard on receipt of data. Note that a two-state guard has not been assumed. Lowering could well be a progressive process. Indeed, the functions "raised" and "lower", thus far, have no semantic meanings, our description being purely syntactic.

Next, we declare a piece of software to implement the specification:

```
[LEVEL(1) PROCSPECS]

PROCSPEC prepare(IN d: data
                 INOUT g: guard)
DERIVES g AS lower(g, d)                                    (3.3)
PRE     raised(g)
POST    g = lower('g, 'd);
```

In IL, neither functions nor operators have bodies. They represent mathematical objects, for example the specifications of pieces of software; but they do not represent software implementations which must be modelled by main programs and procedure bodies.

The task of procedure "prepare" is to implement the function "lower". Thus "prepare" changes the state of the guard by "lowering" it. Initially, the guard is to be in the "raised" state. In the post-condition, primed variables denote initial states, unprimed variables final states.

Note that "raised" and "lower" are used in the specification of "prepare". In a sense, they are proof functions. At a later stage, the body of "prepare" will be compared against its specification. For the present, a trivial body is supplied by using the specification (3.3) directly.

```
[TRIVIAL LEVEL(1) PROC BODIES]

PROC prepare;                                               (3.4)
  g := lower(g, d)
ENDPROC;
```

Concatenating the pieces (3.2), (3.3) and (3.4) and appending the word FINISH yields a syntactically correct IL program purporting to represent the top-level requirement. The program may be verified correct trivially via the Compliance Analyser.

At this stage we must turn to the customer who will either respond in appreciation of what has been produced or will notice that nothing has been said of warning lights. Perhaps "prepare" should also indicate the conclusion of the safety-assessment. The procedure would then require more parameters; but such a modification is better made early rather than late.

Assuming that agreement has been reached at level(1), suppose that a further conversation takes place to refine (3.1) yielding the following proposals:

```
the guard is to comprise two safety-breaks;
each break is to identify the driver
and assess the safety of the car                        (3.5)
before allowing the car to start;
unauthorised drivers are to be inhibited.
```

The intention is to inject redundancy into the system. Only if both safety-breaks are satisfied will the car be allowed to start. Together, they will need to assess the safety of the car (states of fluids, electric circuits, tyres, brakes etc) and check that the driver is authorised. The requirement (3.5) is by no means deterministic, many solutions being possible. For example, the two breaks could be identical or designed to check different parts of the car. An expression of (3.5) in IL that leaves open these options now follows.

```
[LEVEL(2) SPECS]

TYPE break, datum;
REFINED TYPE guard = RECORD b1, b2: break ENDRECORD;
REFINED TYPE data = RECORD d1, d2: datum ENDRECORD;

FUNCTION open(break): boolean;
FUNCTION close(break, datum): break;
                                                        (3.6)
REPLACE (g: guard)
raised(g) BY
open(B1 g) AND open(B2 g);

REPLACE (g: guard; d: data)
lower(g, d) BY
make_guard( close(B1 g, D1 d), close(B2 g, D2 d) );
```

The level(2) specifications begin by stating that a guard comprises a pair of safety-"breaks" while the "data" received by guard comprises a pair of "datum(s)" (!). The function "close" is to change the state of a break by an amount depending on the datum received while "open", analogous to its higher level counterpart "raised", decides whether a break lies in its specified initial state. Note that "open" and "close" as yet have no semantic meanings. The rewrite rules, heralded by the key-word REPLACE, attach semantic meanings to the level(1) functions "raised" and "lower" in terms of "open" and "close".

Thus the use of REFINED TYPE and REPLACE establishes the interface between the specifications at levels (1) and (2). In short, the level(1) types and functions have now been defined in terms of their level(2) counterparts.

To implement the level(2) specifications we declare a procedure:

```
[LEVEL(2) PROCSPECS]

PROCSPEC withdraw(IN d: datum
                  INOUT b: break)                       (3.7)
DERIVES b AS close(b, d)
PRE     open(b)
POST    b = close('b, 'd);
```

The purpose of procedure "withdraw" is to "close" a "break", initially in the "open" state, on receipt of "datum". Again, there is no assumption that a break is bi-state.

Since the procedures "prepare" and "withdraw" implement the functions "lower" and "close" and since a replacement rule defines "lower" in terms of "close", there is now a semantic connection between "prepare" and "withdraw". This connection must be preserved by their bodies. Previously, we provided a trivial body (3.4) for "prepare". We now substitute a refinement:

```
[LEVEL(1) PROC BODIES REFINE ]

PROC prepare;
VAR b1, b2: break;
  b1 := B1 g;
  b2 := B2 g;                                          (3.8)
  withdraw(D1 d, b1);
  withdraw(D2 d, b2);
  g := make_guard(b1, b2);
ENDPROC;
```

The refinement of "prepare" is straightforward. First, the guard is decomposed into its constituent safety-breaks. Next, the level(2) procedure "withdraw" is called twice, once for each break and with the datum appropriate to that break. (These could equally have been implemented in parallel using the constructs MAP and ENDMAP.) Finally, the modified breaks are reassembled metaphorically so producing the guard in one of its lowered states.

To complete level(2), a trivial body for "withdraw" is provided, using the specification (3.7) directly:

```
[TRIVIAL LEVEL(2) PROC BODIES]

PROC withdraw;                                         (3.9)
  b := close(b, d)
ENDPROC;
```

Concatenating the pieces (3.2), (3.3), (3.6), (3.7), (3.8) and (3.9) and appending the word FINISH produces an IL text that may be verified using the Compliance Analyser. This means that the system at level(2) is a correct refinement of that at level(1).

Assuming customer-agreement, we proceed to level(3):

```
[LEVEL(3) SPECS]

TYPE switch, lock, safe_key, auth_key, angle;
REFINED TYPE break = RECORD sw: switch; lo: lock ENDRECORD;
REFINED TYPE datum = RECORD sk: safe_key; ak: auth_key
                     ENDRECORD;

FUNCTION off(switch): boolean;                          (3.10)
FUNCTION match(safe_key, auth_key, lock): angle;
FUNCTION turn(switch, angle): switch;

REPLACE (b: break)
open(b) BY
off(SW b);

REPLACE (b: break; d: datum)
close(b, d) BY
make_break( turn( SW b, match(SK d,AK d,LO b) ) , LO b );
```

The evolution of the design from level(2) to level(3) is analogous to that from level(1) to level(2). The refinement of types is presented first. It is stated that a "break" comprises a "switch" and a "lock" while "datum" consists of a pair of keys, a "safe(ty)-key" and an "auth(orisation)-key". (By a lock here is m ant the female counterpart to a key or keys. Effectively, locks are constants, the mechanism being part of the switch.) The two keys represent respectively the safety-state of the car and the identification-number input by the driver immediately prior to attempting to start the car.

The declaration of the intermediate type "angle" is to aid the readability of the text but is otherwise inessential.

The functions "off" and "turn" are analogous to "open" and "close" at level(2). The intermediate function "match" produces an angle from the two keys comprising datum and from the lock embedded in the break.

The rewrite rules serve to define the level(2) functions in terms of their level(3) counterparts. (Of course, the latter are still semantically undefined.) In particular, closing a break is defined by turning a switch through an angle given by matching the two keys with the lock.

The software to implement the level(3) specifications is now declared:

```
[LEVEL(3) PROCSPECS]

PROCSPEC rotate(IN sk: safe_key
                IN ak: auth_key
                IN lo: lock
                INOUT sw: switch)                          (3.11)
DERIVES sw AS turn( sw, match(sk,ak,lo) )
PRE     off(sw)
POST    sw = turn( 'sw, match('sk,'ak,'lo) );
```

The purpose of "rotate" is to implement the behaviour of switch under the influence of "match" and "turn"; and its declaration enables us to refine the level(2) procedure body (3.9).

```
[REFINED LEVEL(2) PROC BODIES]

PROC withdraw;
VAR switch: switch;
  switch := SW b;                                          (3.12)
  rotate(SK d, AK d, LO b, switch);
  b := make_break(switch, LO b);
ENDPROC;
```

The design of the body of the level(2) procedure "withdraw" follows that of "prepare". The break is decomposed into its switch and lock. The level(3) procedure "rotate" is then called. The switch is rotated through an angle determined by the keys and lock. Finally, the break is reassembled. Henceforth, (3.12) replaces (3.9).

We conclude by providing a trivial body for "rotate" by equating it to its specification (3.11):

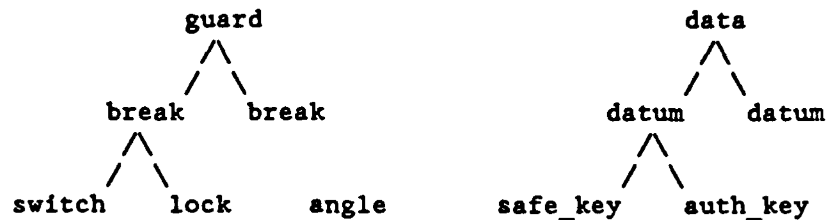
```
[TRIVIAL LEVEL(3) PROC BODIES]

PROC rotate;                                                  (3.13)
sw := turn( sw, match(sk,ak,lo) );
ENDPROC;
```

Of course, further refinement would replace this body by one referring to declarations at a lower level. Indeed, further refinements might be necessary prior to implementation in say Ada, Pascal or Coral.

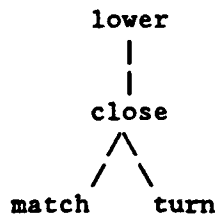
For brevity, only three levels in the hierarchy have been presented and these may be depicted graphically:

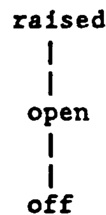
DATA TYPES



(3.14)

FUNCTIONS





PROCEDURES

Data types at different levels in the hierarchy are related via the projection and construction functions implicit in the declarations of the records that fulfil the refinement process. Functions at different levels are related by means of user-defined rewrite-rules while the procedure hierarchy is that of the call graph.

Finally, the design is easily and speedily verified using the MALPAS Compliance Analyser. This is not surprising because the proof of correctness was built into the design from the very start.

For ease of examination, the components of the system are presented together in the appendix.

4 CONCLUDING REMARKS

The purpose of this paper has been to outline some preliminary thoughts on the development of verified software using a hierarchy of descriptive levels in the hope that this may provide a practical, cost-effective method of producing large systems of high quality. Indeed, a method is evolving that seeks to satisfy a major principle that has so far failed to find serious expression, namely:

Before development commences, (4.1)
the customer does not know in detail
what he wants.

It seems unlikely that the traditional sequence of software procurement involving requirements capture, feasibility study, project definition, design and implementation, is sufficiently flexible to account for the principle (4.1). Indeed, for a given system, the method advocated here would imply a top level description almost devoid of semantic detail. Meaning is infused via a process of successive refinement with the specification and design proceeding together.

There are two issues for debate. First, there are those (eg [7]) who question whether development really can occur in a rational, top-down manner. At any rate they agree that a faithful top-down description, however produced, is desirable for purposes of understanding, certifying and maintaining the software. Even if derived post facto, the demand for such a description imposes severe and beneficial constraints upon the production method.

Secondly, there is the choice of language. There are arguments for the separation of the languages of specification, design and implementation on the grounds that the "what", being fundamentally different from the "how", requires a different style of expression. Although there are differences, implementations being efficient forms of specifications, I have not found such arguments compelling. Certainly, a language to cover both the "what" and the "how" has to be more expressive than a standard programming language. At the same time, it should not be so esoteric that its use is restricted to a handful of computer science graduates.

In this paper, I have presented a semi-formal specification and design method based upon a hierarchy of descriptive levels. It is to be hoped that more formality will emerge once experience has been gained with real examples. In particular, a generalisation of the algorithm (1.1) is needed that will cater for systems whose natural description is recursive in nature. This would generate graphs, akin to (3.14), but containing cycles. Also, if types can be refined in terms of types declared later or at the same level, can they be refined using types declared earlier?

Finally, it is worth noting that the method presented here, though targeted at software development, may have a much wider application. Indeed, it may be the gateway to a method for producing the correct designs of quite general systems.

REFERENCES

1 B D Bramson:
Tools for the specification, design, analysis and verification of software; RSRE report 87005.

2 P Henderson:
Functional programming, formal specification and rapid prototyping; IEEE Trans Soft Eng SE-12, 2, 241-250, 1986.

3 L C Carpenter & L L Tripp: Software design validation tool; ACM Sigplan Notices 10, 6, 395-400, 1975.

4 H F Ledgard: The case for structured programming;
BIT 13, 45-57, 1973.

5 P G Neumann:
On hierarchical design of computer systems for critical applications;
IEEE Trans Soft Eng SE-12, 9, 905-920, 1986.

6 B Billard: Scenarios for MALPAS applications; RSRE report 86015.

7 D L Parnas & P C Clements:
A rational design process: how and why to fake it;
IEEE Trans Soft Eng SE-12, 2, 251-257, 1986.

APPENDIX: THE COMPLETE EXAMPLE

```
[LEVEL(1) SPECS]

TITLE pre_start_checks;
TYPE guard, data;
FUNCTION raised(guard): boolean;
FUNCTION lower(guard, data): guard;


[LEVEL(1) PROCSPECS]

PROCSPEC prepare(IN d: data
                 INOUT g: guard)
DERIVES g AS lower(g, d)
PRE     raised(g)
POST    g = lower('g, 'd);


[LEVEL(2) SPECS]

TYPE break, datum;
REFINED TYPE guard = RECORD b1, b2: break ENDRECORD;
REFINED TYPE data = RECORD d1, d2: datum ENDRECORD;

FUNCTION open(break): boolean;
FUNCTION close(break, datum): break;

REPLACE (g: guard)
raised(g) BY
open(B1 g) AND open(B2 g);

REPLACE (g: guard; d: data)
lower(g, d) BY
make_guard( close(B1 g, D1 d), close(B2 g, D2 d) );


[LEVEL(2) PROCSPECS]

PROCSPEC withdraw(IN d: datum
                  INOUT b: break)
DERIVES b AS close(b, d)
PRE     open(b)
POST    b = close('b, 'd);
```

```
[LEVEL(3) SPECS]

TYPE switch, lock, safe_key, auth_key, angle;
REFINED TYPE break = RECORD sw: switch; lo: lock ENDRECORD;
REFINED TYPE datum = RECORD sk: safe_key; ak: auth_key
                     ENDRECORD;

FUNCTION off(switch): boolean;
FUNCTION match(safe_key, auth_key, lock): angle;
FUNCTION turn(switch, angle): switch;

REPLACE (b: break)
open(b) BY
off(SW b);

REPLACE (b: break; d: datum)
close(b, d) BY
make_break( turn( SW b, match(SK d,AK d,LO b) ) , LO b );


[LEVEL(3) PROCSPECS]

PROCSPEC rotate(IN sk: safe_key
                IN ak: auth_key
                IN lo: lock
                INOUT sw: switch)
DERIVES sw AS turn( sw, match(sk,ak,lo) )
PRE     off(sw)
POST    sw = turn( 'sw, match('sk,'ak,'lo) );


[LEVEL(1) PROC BODIES REFINED]

PROC prepare;
VAR b1, b2: break;
  b1 := B1 g;
  b2 := B2 g;
  withdraw(D1 d, b1);
  withdraw(D2 d, b2);
  g := make_guard(b1, b2);
ENDPROC;


[LEVEL(2) PROC BODIES REFINED]

PROC withdraw;
VAR switch: switch;
  switch := SW b;
  rotate(SK d, AK d, LO b, switch);
  b := make_break(switch, LO b);


[TRIVIAL LEVEL(3) PROC BODIES]

PROC rotate;
sw := turn( sw, match(sk,ak,lo) );
ENDPROC;

FINISH
```

DOCUMENT CONTROL SHEET

Overall security classification of sheet ....Unlimited..

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	Memo 4063		Unlimited

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
	RSRE, Saint Andrews Road, Malvern, Worcs WR14 3PS.
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title
An approach to the hierachical specification, design and verification of software.

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
BRAMSON B D			1987.7	11

11. Contract Number	12. Period	13. Project	14. Other Reference
..			

15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

See text

END
Feb.
1988
DTIC