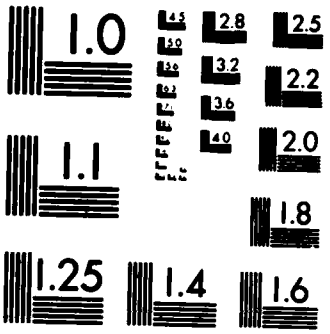
AD-A141 064 SPECIFICATION AND TRANSFORMATION: AUTOMATED IMPLEMENTATION(U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION S.. M S FEATHER APR 84 1/1

UNCLASSIFIED ISI/RS-83-C-0335 F30602-79-C-0042 F/G 9/2 NL



ISI Reprint Series
ISI/RS-83-124
April 1984

Martin S. Feather

University of Southern California

Specification and Transformation: Automated Implementation

Reprinted from *Proceedings of the Program Transformation and Programming Environments Workshop*, Munich, Germany, September 1983.

INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/RS-83-124	2. GOVT ACCESSION NO. AD-A141 064	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Specification and Transformation: Automated Implementation		5. TYPE OF REPORT & PERIOD COVERED Research Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Martin S. Feather		8. CONTRACT OR GRANT NUMBER(s) MDA903 81 C 0335
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90292-6695		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE April 1984 13. NUMBER OF PAGES 10
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* ----------		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

This document is approved for public release; distribution is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

----

18. SUPPLEMENTARY NOTES

This report is a reprint of a paper that appears in the proceedings of the Program Transformation and Programming Environments Workshop, held in Munich, Germany, in September 1983. The proceedings were published by Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, USA.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

automated implementation, program specification, program transformation

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. ABSTRACT

~~Our~~ research group at ISI aims to improve the program development process by applying ***program transformation*** to develop implementations from specifications.

The uses for specifications imply the criteria for a specification language -- understandability, testability and maintainability. We have *designed* our own specificaiton language, ***Gist***, to meet these criteria. We have sought inspiration for Gist's constructs from the power found in descriptions expressed in natural language.

The justifications for our approach are outlined. Our experiences with specification in Gist, and subsequent transformation of such specifications, suggest some implications for the processes of specification and transformation. References point the way to more details on the various issues.

ISI Reprint Series
ISI/RS-83-124
April 1984

University of Southern California

Martin S. Feather

Specification and Transformation: Automated Implementation

Reprinted from *Proceedings of the Program Transformation and Programming Environments Workshop*, Munich, Germany, September 1983.

INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

This research is supported by the Defense Advanced Research Projects Agency under Contract No. MDA903 81 C 0335. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government or any person or agency connected with them.

ISI Reprint Series

This report is one in a series of reprints of articles and papers written by ISI research staff and published in professional journals and conference proceedings. For a complete list of ISI reports, write to

Document Distribution
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695
USA

SPECIFICATION AND TRANSFORMATION: AUTOMATED IMPLEMENTATION

Martin S. Feather
Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
U.S.A.

The primary aim of our ISI research group[1] is to improve the software development process. We contend that the foremost means of achieving improvement is to significantly expand the role of the computer in the process, and advocate program transformation as a development methodology within which to introduce such computer support. As evidenced by the other contributions in this volume, we are not alone in this belief.

The program transformation methodology is characterised by the machine acquisition of a *specification*, which is then *transformed*, with human assistance, into a program. Notice the distinction between this and conventional programming, where it is the *program* that is acquired. The strength of the transformational approach derives from the freedom to write specifications in a language for which automatic compilation need not be guaranteed. This freedom strongly affects both the specification language design and the methodology itself: there are no *a priori* limits to imaginable language constructs and the transformation system can become a cooperative partner in the design process, unlike batch compilation mechanisms. This freedom allows recording even the early stages of the software development process and should ultimately facilitate their automated support.

Research efforts in program transformation are centered on designing the specification language and building tools and techniques around that language to support specification acquisition and

---

[1]The work reported herein has been sponsored by the Defense Advanced Research Projects Agency, Contract # MDA903 81 C 0335, the National Science Foundation, Contract # MCS-7918792 , and the Air Force Systems Command, Rome Air Development Center, Contract # F30602-79-C-0042. Views and conclusions contained in this report are the author's, and should not be interpreted as representing the official opinion or policy of DARPA, NSF, RADC, the U.S. Government, or any person or agency connected with them.

This report is summary in nature, and represents the work of many individuals. Our group is headed by Bob Balzer. Its present members are: Bob Balzer, Don Cohen, Martin Feather, Neil Goldman, Jack Mostow, Bill Swartout and Dave Wile. Former members who have made significant contributions to this research are: Wellington Chiu, Lee Erman, Steve Fickas, Monica Lam and Phil London.

transformation. As experience accumulates, support for the processes may be added incrementally. While the accumulation of enough transformation techniques may ultimately permit automatic compilation for constructs presently regarded as purely specificational, the transformation process is *inherently* interactive and must always give the user the design initiative.

Here I concentrate upon the specification research of our group, emphasizing how the freedom from compilation has influenced the specification language design and some of the ramifications for the development process itself. The companion paper by Wile focuses upon the development process, in particular, the need for, and means to, record and support as much as possible of all aspects of development activity.

Specification Language Desiderata

Abandoning automatic compilation as a requirement on the specification language allows much more freedom in the design of the language, for the designer can include constructs with multiple (imagined) implementations whose best implementation may be very context dependent. Of most importance, the language design can be based on how it will be *used* rather than how it can be *implemented*.

Uses imply criteria

The foundations of our specification language, Gist, are discussed in some detail by Balzer and Goldman [1]. Again, the uses of specifications determined the design criteria. Briefly, these uses and the criteria that they determine are:

- A specification serves as a *contract* between specifier and implementor defining the system to be constructed. Hence a specification must be *understandable*.
- A specifier must be able to *validate* that the specified system meets the needs for which it was designed. Hence a specification must be *testable*.
- A contract (between specifier and implementor) will *change* over time. Hence a specification must be *maintainable*.

Balzer and Goldman go on to derive broad principles and specific implications for a specification language to meet these criteria.

Inspiration from natural language

The principles and implications do not themselves define a specification language, but rather describe properties desired of such a language. Our group has sought inspiration for what constructs to incorporate into our specification language from the study of *natural language* descriptions. Early experience with a program to automatically acquire formal specifications from natural language convinced us that some of natural language's expressiveness and elegance for modeling should be formalised [3]. Since our aim is to construct a formal specification language, inspiration is drawn less from the informality or ambiguity of natural language than from the rich

modes of expression that have readily formalisable counterparts. Goldman and Wile [6] follow this approach in designing Gist, our group's specification language.

Why natural language is a good inspirational source

We have found natural language to be a valuable source of inspiration. We attribute this to the importance of minimising the gap between informal intent and some formalisation of the same. As stated by Goldman and Wile, "In general, a software system is intended to represent the activity in some 'ideal world', which may be an abstraction of a real world process, a purely mental conception of the desired behavior, or a combination of the two." Thus an unbridgeable cognitive gap will always exist between intended ideal and formalisation (i.e., specification) of the same. At best, that gap can only be narrowed, not eliminated. The three criteria for a good specification -- *understandability*, *testability* and *maintainability* -- are supported by narrowing the cognitive gap.

Narrowing may be achieved by populating the specification language with constructs that are formalisations of natural language constructs. This mirroring of the expressive means used to convey informal intent enhances the *understandability* of the specification. The major expressive constructs that have formal equivalents in Gist are: descriptive reference ("the largest message in my message file"), historical reference ("the last message I received"), nondeterminism ("send this message by any route which will deliver it") constraints ("never send the same person more than one copy of a message") and demons ("when I receive a message, notify me").

The specification describes the behaviour of both the program and its environment; this provides a natural way to specify embedded programs that interact in complicated ways with their environment. This, and the expressive constructs, enhance the *testability*, since the specification forms a direct model of the domain.

The aspects of the required functionality are specified merely by stating them, no matter how complex they are, or how many of them need to be integrated together to accomplish this functionality (e.g., the conjunction of nondeterminism and constraints denotes those and only those out of the range of nondeterministically described behaviours that satisfy the constraints). The separate and single statement of each aspect of the functionality enhances the *maintainability* of the specification.

Some experiences with Gist as a specification language

Our experience with Gist has not always reflected our expectations. The following are some of the surprises that have general significance. It is interesting that each can be viewed as the result of being too far from natural language.

Incomprehensibility of formal specifications

Although Gist has been *designed* to be a specification language, formal specifications written in it, like those in all other formal specification languages, tend to be hard to understand. Swartout [8] identifies several reasons for this unreadability.

Two such reasons are the unfamiliar syntax and the lack of redundancy in the specification. His solution to these problems has been to develop a tool that *paraphrases* Gist specifications in English. This has been found to be useful in both clarifying specifications and revealing specification errors. Even experienced Gist users, presumably familiar with the syntax, found the tool helpful for locating errors. Swartout suggests that this is because the English paraphrase gives the specifier an alternate view of the specification which highlights aspects that are easily overlooked in the formal Gist notation.

Swartout's paraphraser deals only with the static aspects of a specification. Also important are the dynamic aspects - the behaviours denoted by a specification. The nature of Gist is such that there are implicit remote interactions between parts of the specification, which are often not apparent from a casual examination of the specification. To reveal these non-local interactions, Cohen [4] is constructing a *symbolic evaluator* for Gist. The symbolic evaluator produces an execution trace which details everything discovered about the specification during evaluation. Since this trace is rather detailed and low-level, i.e., hard to understand, Swartout [9] is building a *trace explainer* that selects those aspects of the trace believed interesting or surprising, and summarises them in English.

Symbolic evaluation and prototyping (rapidly developing a crude implementation) are both valuable means for *validating* a specification, i.e., increasing confidence that the formal specification is the desired specification. The specifiers can gain confidence that the specification matches their informal intent, and the intended users (of the software to be developed from the specification) can gain confidence that the formally specified system will actually meet their needs.

Large specifications

Large size, as well as the other reasons discussed above, can make a formal specification difficult to understand. Goldman [5] suggests that we should look to natural language descriptions to find the means to overcome this particular problem. He finds that descriptions of large systems incorporate an evolutionary vein - the final description can be viewed as an elaboration of some simpler description, itself the elaboration of a yet simpler description, etc., back to some description deemed sufficiently simple to be comprehended from a non-evolutionary description. Goldman proposes that formal specifications likewise be described in an evolutionary vein. Our hope is that the evolutionary steps can be formalised -- i.e., that a language of change can be developed that permits a formal specification to be viewed and analyzed from its evolutionary perspective. Our motivations for recording and supporting the software development process imply that we should record and support this evolution.

Readers interested in details of Gist are referred to [7].

Transformation of Gist

In choosing design criteria for Gist, we were biased totally in favour of specification, making no allowance for the need to transform Gist specifications into programs. Thus although we may have minimised the conceptual gap between intent and formal specification, in doing so we have

maximised the transformation gap between specification and program. This is the "price" of freedom from compilation. We do, however, have confidence that formal Gist specifications lie on the direct route from intents to programs.

Mapping Gist into conventional implementation constructs

Our research strategy for determining how to transform Gist specifications is to focus upon Gist's expressive constructs. The common characteristic of the different constructs is that they provide a means of expressing desired behaviour without prescribing a particular algorithm to achieve that behaviour. It is this freedom from implementation concerns that underlies their success as specification constructs. Balzer, Goldman and Wile [2] describe in some detail the nature of these freedoms and how they are provided by Gist's constructs. The implementation concerns fall into three broad categories: finding a *method* for accomplishing something, providing the *data* required for that method, and making the combination *efficient* (some function of time, space and other resources). Implementation is a matter of introducing these implementation concerns.

For each construct, our research aims to accumulate the following:

- *implementation options*, commonly available options for converting an instance of the specification construct into a more efficient expression of the same behaviour, typically in terms of lower level constructs;
- *selection criteria*, for selecting among several implementation options applicable to the same instance; and
- *mappings*, to achieve the implementation options via sequences of correctness preserving transformations.

These techniques are applied during the transformation of a Gist specification into a program, by focusing upon the instances of specification constructs. For each given instance, the applicable implementation options are identified, the selection criteria are applied to suggest the appropriate option, and the corresponding mapping applied to effect the implementation by means of transformations. Further details in this regard are presented in [7].

An automated, not automatic, activity

Developing a program from the specification is by no means a fully automatic activity, nor will it become one in the near future for any but the most trivial of specifications. There are several important reasons that this should be the case:

- *lack of coverage*: the techniques that we have accumulated fall within a broad spectrum, from general purpose techniques that are generally applicable to a range of instances of a construct, to special purpose techniques that apply to some idiomatic use of a construct. There will be occasions where general purpose techniques do not result in sufficient efficiency, and none of the special purpose techniques are applicable to the particular instance.

- *implementation lag*: our implementation of the techniques lags behind our discovery of them.

- *interactions*: implementations of several instances of constructs in a specification are not necessarily independent; in particular, the optimal implementation of several instances need not be the combination of the optimal implementations for those instances considered separately.

- *local optimisation*: typically, the mapping of a specification construct into an implementation results in the distribution of code throughout the program to achieve that same behaviour. These pieces of code may often be simplified in the local contexts in which they are deposited. Not all such simplification will be within the capability of an automatic simplifier.

- *feedback*: discoveries made in the course of attempting implementation may suggest modifications to the specification.

These reasons combine to make transformation an *interactive* process, between skilled implementor and some supporting computer system. The nature of this support is the fo of the companion paper by Wile. Briefly, the overall objective is to streamline the implementor's ; one of decision making and guidance, leaving the system to perform the activities of man, ation, analysis and documentation.

There are some important consequences for the transformation process that I would like to emphasize:

- *wide-spectrum language*: the intermediate stages, between specification and program, must be expressed in some language. Preferably that language should be capable of mixing specification and implementation constructs (since otherwise the transformation process would have to be stratified into many separate stages). We have found that Gist is appropriate as a wide-spectrum language, at least for the earlier stages of transformation (our explorations have not descended to very low level algorithmic details). The transformation and analysis tools must be capable of operating on the mix of constructs. Thus there is a tradeoff between mixing and stratification: the former admits a more continuous and flexible transformation process, whereas the latter simplifies the tool building by requiring that the tools deal only with the mix of constructs to be found in each layer.

- *view intermediate stages*: the implementor must be able to view those intermediate stages during the transformation process. Thus applying transformations must result in intermediate stages that are not only meaningful, but also readable and comprehensible by the implementor. Again, stratification of the transformation process might have some virtue, since then the implementor would not have to consider arbitrary mixes of constructs from different layers.

- *switch to compilation*: despite our emphasis that the transformation process cannot be an automatic process, we recognise that at some point in the transformation process, uses of specification constructs will have been eliminated or constrained sufficiently to permit automatic compilation into a tolerably efficient program. Just when this stage has been reached will be determined on a case-by-case basis. Particularly stringent efficiency

requirements may force further transformation of the intermediate specification before applying automatic compilation. Notice that although compilation is automatic, input may be more than just the intermediate code. It may include additional information on how to do the compilation (e.g., directions on data structure selection). This tends to blur the distinction between compilation and transformation, although the expectation is that compilation, once begun, is automatic (i.e., all the pertinent information is provided at the start, the process requiring no intervention later on) and the final step (i.e., the implementor will not modify the result of compilation). Our own research efforts have identified a subset of Gist (called Will), which represents our state of knowledge of what we know how to compile. It is expected that this subset will expand as our knowledge increases.

References

1. Balzer, R. & Goldman, N., "Principles of good software specification and their implications for specification languages," in *Specification of Reliable Software*, pp. 58-67, IEEE Computer Society, 1979.

2. Balzer, R., Goldman, N. & Wile, D., "Operational specification as the basis for rapid prototyping," *ACM Sigsoft Software Engineering Notes* 7, (5), December 1982, 3-16. Working papers from the ACM SIGSOFT Rapid Prototyping Workshop

3. Balzer, R., Goldman, N. & Wile, D., "Informality in program specifications," *IEEE Transactions on Software Engineering* SE-4, (2), 1978, 94-103.

4. Cohen, D., "Symbolic execution of the Gist specification language," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany*, pp. 17-20, August 1983.

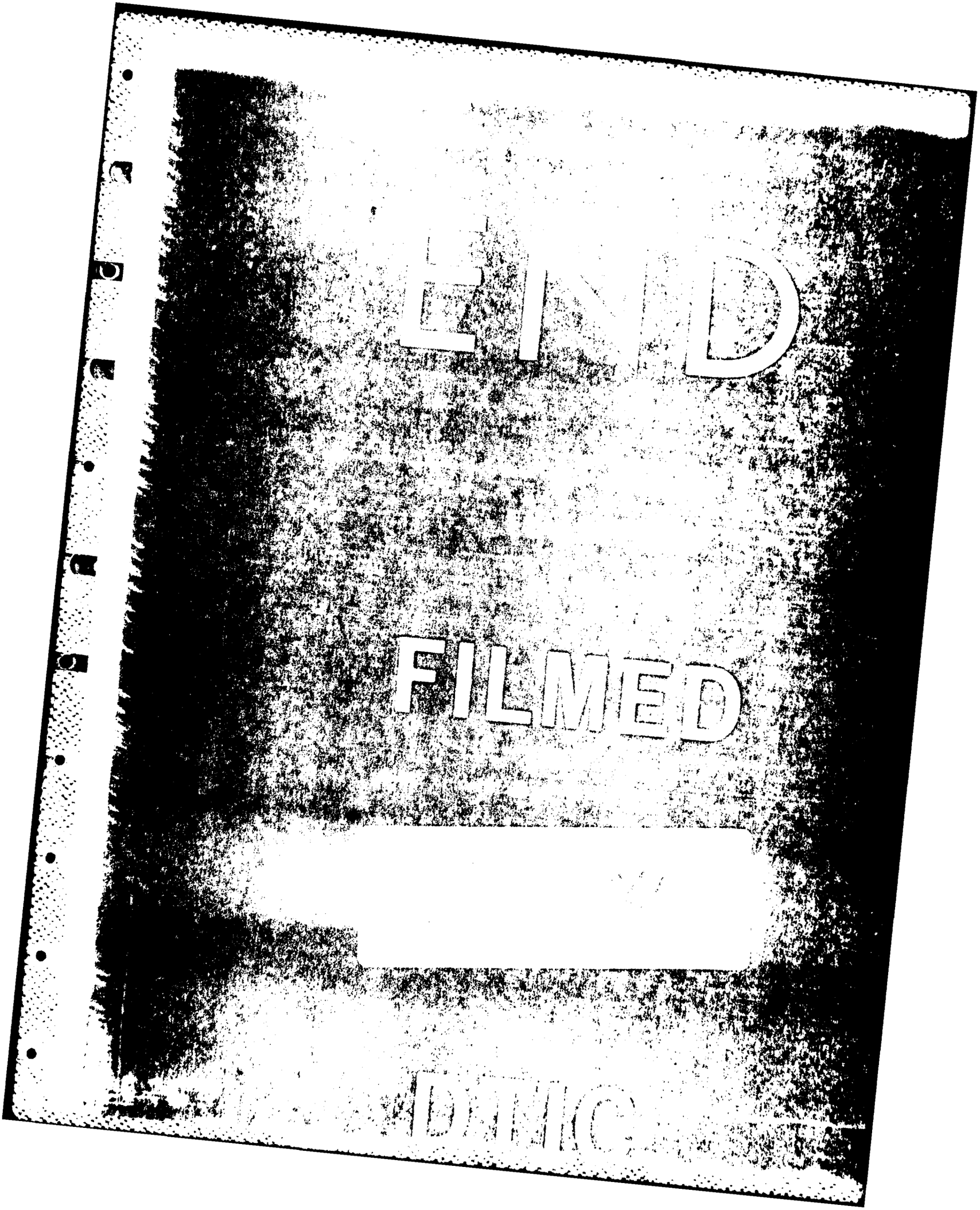
5. Goldman, N.M., "Three dimensions of design development," in *Proceedings, 3rd National Conference on Artificial Intelligence, Washington D.C.*, pp. 130-133, August 1983.

6. Goldman, N. & Wile, D., "A relational data base foundation for process specification," in Chen (ed.), *Entity-Relationship Approach to Systems Analysis and Design*, pp. 413-432, North-Holland Publishing Company, 1980.

7. London, P.E. & Feather, M.S., "Implementing specification freedoms," *Science of Computer Programming*, (2), 1982, 91-131.

8. Swartout, W., "Gist English generator," in *Proceedings, AAAI-82*, pp. 404-409, August 1982.

9. Swartout, W., "The GIST behaviour explainer," in *Proceedings, 3rd National Conference on Artificial Intelligence, Washington D.C.*, pp. 402-407, August 1983.


END
FILMED
DTIC