Verification of ANDF components

Frédéric Broustaut, Christian Fabre, François de Ferrière, Éric Ivanov – Open Software Foundation Research Institute[1].

Mauro Fiorentini – Etnoteam[2].

This paper presents validation work[3] done on ANDF at the Open Software Foundation Research Institute. The ultimate ANDF scenario splits a compiler into two separate components (producer/installer). This changes the compiler validation process as the two components have to be validated separately. This paper presents the originality and the difficulties of such an approach and summarizes the status of two pieces of software to which the OSF-RI has contributed: the *ANDF Validation Suite* and the *General ANDF Interpreter*[4].

1. Introduction

ANDF is an Architecture- and language-Neutral Distribution Format developed by OSF and other collaborators around the world[5]. It is based on an Intermediate Language (IL) with specific features to address software portability issues. The goal of ANDF is to create an Architecture Neutral representation of programs which may then be installed right out of the box on a variety of target architectures.

1. OSF Research Institute, 2 avenue de Vignate, 38610 Gières, France. `andf-ri@osf.org`, Fax: +33 76-51-05-32, Tel.: +33 76-63-48-64.
2. Etnoteam S.p.A., Via Adelaide Bono Cairoli 34, Milan 20127, Italy. `mfiorentini@etnoteam.it`, Fax: +39 2 261-107-55. Tel.: +39 2 261-621.
3. This work has been partially funded by the CEC under the project OMI/GLUE (Esprit project n°6062). For more information, please contact Gianluigi Castelli, Etnoteam S.p.A. (see address above).
4. The General ANDF Interpreter has been developed in collaboration between the OSF-RI and Etnoteam. See § 10 for details.
5. It is based on the TDF technology provided by the Defence Research Agency of the United Kingdom Ministry of Defence. DRA is prepared to assist vendors with the industrialization of TDF. Interested parties should contact DRA directly at Defence Research Agency, St. Andrews Road, Malvern, Worcestershire, WR14 3PS, England. Tel.: +44 1-684-89-53-14.

Structure of the document

The rest of the document is organized as follows:

- A summary about ANDF is presented in the "*Experiments with ANDF*" (see § 2) and "*The ANDF scenario*" (see § 3) presents the final goal of an Architecture Neutral Distribution Format, how it impacts the compilation process and how portability is achieved.
- The technology is presented in "*ANDF features*" (see § 4), which describes the features of the technology, and the "*ANDF components*" (see § 5) are then introduced.
- After an introduction to "*Validation of ANDF components*" (see § 6), the "*ANDF Validation Suite*" (see § 7) and the "*General ANDF Interpreter*" (see § 8) are presented, as well as their interaction in "*The AVS and the GAI*" (see § 9).
- The context of this work as well as "*Related work*" (see § 10) are outlined, and the paper ends with "*Conclusion and Further work*" (see § 11).

2. Experiments with ANDF

DRA's TDF technology, chosen by OSF to become its ANDF, is a very versatile piece of software. Up to now, the technology has been extensively used and exercised under different Unixes. A feasibility study has also been carried-out for the Window environment, more precisely on the Win32 Application Programming Interface (API) [10]. The experiments carried-out so far have shown a wide spectrum of potential applications:

- The most basic point of view, is to see it –as any other intermediate language– as a *vehicle for the construction of modular compilers.* While this scenario allows separate developments of the front-end and back-end, and to a greater extent than that of other ILs, the resulting compiler is still meant to be used as an integrated tool. A typical case in this scenario is to use DRA's `tcc` as the default development compiler on a multiple platform environment.
- One step further, the technology has proven a very useful tool in the quest for improved portability of application programs: it is then used as a *portability checker.* The TDF tools have a substantial capability to highlight port-

ability issues, such as adherence to, and proper use of, standard languages and standard APIs. One of the main results of OSF experiments is that most of the issues raised when using the technology concern discrepancies among different Unix implementations, rather than ANDF issues as such [9]. Recent initiatives in the Unix world, such as *spec-1170*[1], will reduce such conflicts in the near future.

- At the other end of the spectrum is the scenario of the *Architecture Neutral Distribution Format*. This scenario is described § 3, but the idea is that the end-user buys a software package and installs it on his platform through a completely automatic process, regardless of any architectural or language details.

ANDF will bring no magic: an application will have to be intrinsically portable before ANDFizing it could make any sense, if doable at all. Recent studies have shown that many ISVs have a hand-crafted approach to portability. Usually they develop their software on one platform and port it to other platforms later on, rather than addressing portability from scratch through the use of standard APIs. The current status of the industry is such that portability is addressed as something that has to be lived with, rather than as a competitive advantage.

Thus, the ultimate goal of ANDF, will be achievable only when ISVs start to address portability earlier on in their engineering process through the use of standard APIs and API checkers. In other words, an Architecture Neutral Distribution Format is viable, but the industry approach to portability will also need to evolve.

3. The ANDF scenario

A sensible porting strategy is to write an application which uses standard compliant APIs on a platform where a validated compilation chain provides support for those standards. In an ideal world, the porting effort to another platform supporting the same set of APIs should be reduced to a single recompilation on the new platform.

Abstract APIs

The figure below shows how this changes with ANDF. Our example application uses the spec-1170 API for the system services and the Motif[2] API for the GUI.

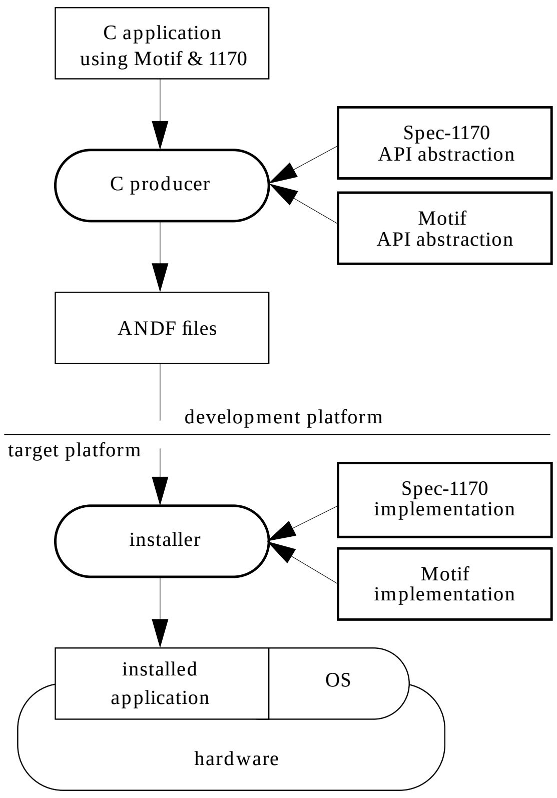
Tokens, together with a *unique* naming mechanism, are used to provide abstractions of APIs. *E.g.* the API object `errno` is defined by [8] to be in `<errno.h>` with the following properties:

It is unspecified whether `errno` is a macro or an identifier declared with external linkage.

Therefore:

- The only thing that the C producer will know about `errno` is the fact that it is a *modifiable lvalue* of type `int`. In fact, `errno` might be implemented as an external symbol, a location into a system area accessed from a hidden pointer, *etc.*
- The installer will be provided with the actual definition of `errno` on the target platform.

A fortunate side-effect of the producer knowing very little about an API object, is that it has to check more thoroughly the usage made of this API object in the source code. Consequently, ANDFizing a piece of code will highlight more portability issues than a tra-



1. *Spec-1170* consists of operating system headers and interfaces. They are published by X/Open as *X/Open Portability Guide issue 4.2.*

2. The *OSF/Motif* toolkit has been standardized by the IEEE under the reference *IEEE Standard for Information Technology – X Window System – Modular Toolkit Environment (MTE), IEEE Std 1295-1993.*

ditional compilation will. Consider the following C fragment:

```
#include <errno.h>
extern int errno ;
/* …other code… */
```

It will pass unnoticed under many compilers because it happens that `errno` is actually implemented this way on many platforms. But the C producer will complain because it has been told earlier that `errno` was only a modifiable l-value, which is not necessarily a global variable.

As a matter of fact, just attempting to ANDFize code has proven to be a very useful way of improving the portability of the code in terms of proper use of the language and APIs.

The compilation process is fragmented

As opposed to the traditional porting scheme, where everything is done on the target platform, the correct compilation and execution of an application using ANDF relies on a chain whose elements may never be used together, but more likely at different times and places. *I.e.* the producer will be used at the software developer's site, while the installer will be used at the user's site, possibly months later.

Therefore, a high level of confidence in the ANDF *components* is necessary for the success of ANDF.

4. ANDF features

ANDF is a full programing language situated somewhere between HLLs (High Level Languages) such as Ada, C or Fortran, and low-level ILs such as the RTL of `gcc`. ANDF provides constructs to describe data types, variables and operators.

In addition, portable use of APIs is done through two ad-hoc mechanisms that delay the actual usage until installation on the target platform, while ensuring that the use is correct at the production level. These two mechanisms to tackle portability are: *tokens* and *conditional* constructs.

A more detailed description of ANDF features follows.

ANDF files

The basic ANDF file is called a *capsule*. A capsule can import or export declarations or definitions of ANDF entities by binding them with external names. A set of capsules can be packed into an *ANDF library*[1]. An *ANDF linker* uses the external references to bind together multiple capsules and / or libraries.

1. Often named a *token library* because it is currently mainly used to hold the target-dependent definition of API tokens.

Data types

ANDF *shapes* describe the basic data types of programming languages: *integers*, *floating-point* numbers (including *complex* numbers), *bitfields*, *pointers*, *offsets* and *procedures*. It is also possible to recursively build *structures*, *unions* and *arrays*. These types can be parameterized by different means:

- Integers, floating-point numbers and bitfields are parameterized by the range of values they can represent.
- Pointers and offsets are characterized by their *alignment*, *i.e.* the memory layout of the data they refer to.
- Tokens and conditionals can be used to parameterize shapes. This allows parameterized data types to be built that depend on features of the target architecture.

Operators

The ANDF language defines a set of constructs, about one hundred, for arithmetic and pointer operations, memory object access, and flow control. They have been designed to allow a straightforward translation from a large set of high level language constructs, and to permit easy implementation on most architectures. In addition, some of these constructs can hold information for specific behavior, such as overlay of memory on assignment, or for optimization purposes such as profiling information on conditions. For arithmetic operations, specific treatments can be set-up if an exception occurs.

Variables & Identifiers

The basic variable identifier in ANDF is a *tag*. A tag is referred to by an integer. An integer refers to only one tag. However, a given tag can not be used everywhere: it has got a scope of visibility.

A tag can be given an *external name* visible outside the capsule.

Tokens

A token is a parametrized placeholder for actual ANDF code. Most pieces of ANDF could be tokenized.

When the declaration of a token is known, it may be used in relevant places as plain ANDF code. When the definition of the token is actually provided, references to it are expanded.

Conditionals

A conditional construct is just an integer test expression with two branches. The expression is evaluated at installation time, and according to its result, one of the two branches is selected and the remaining branch is discarded. Note that the integer expression might depend on tokenized values.

5. ANDF components

Two typical components in the ANDF scenario are producers, that compile code from an HLL into target independent ANDF, and installers that translate this ANDF code into object code.

The production process is very HLL dependent. The main role of the producer being to check the validity of the source code, and then to generate ANDF.

The installation process can be logically divided into the following phases:

Token binding, where the definition of target dependent tokens are provided,

Conditional expansion, where the conditional constructs are evaluated and replaced by one of their branches. At this point we obtain target dependent ANDF.

Translation, where the target dependent ANDF is translated into object code. This phase only performs target code generation, as all the target dependent information has been provided by the expansion of tokens and conditionals.

Linking, where the system linker is used to create the final executable from the object files.

6. Validation of ANDF components

The fact that the compilation process is now fragmented, with the two components knowing nothing about each other has many implications:

- The producer and the installer must be validated separately, whereas in the traditional scheme, the front-end and the back-end of a compiler are validated together on a single platform.
- The user's platform must provide a conformant implementation of the APIs used by the application, Motif and spec-1170 in our example, whereas in the traditional porting scheme, final fine-tuning or fixing of a discrepancy for a given platform is always possible.
- The installer must have access to all the system tools necessary to install a software package without any intervention from the user, *e.g.* system libraries.

It is worth noting that validation of ANDF components by only means of tests suites for a specific language is inadequate, for a number of reasons:

- Each language is likely to generate code with a specific profile: *e.g.* a Fortran producer is more likely to pass parameters by reference only, thus never exercising the pass-by-value convention.
- Producers might make undocumented assumptions about the target architecture; *e.g.* a C producer might assume that:
  `sizeof(long) == sizeof(int).`
- Producers and installers might share a number of unspecified assumptions that would not be found by testing them together with an HLL suite.

Therefore, the approach chosen for validation is as follows:

- Installers are validated with specific test suites: the "*ANDF Validation Suite*" (see § 7). It is a collection of hand-written test cases that exercise each ANDF construct in turn.
- A "*General ANDF Interpreter*" (see § 8) (GAI) has been designed to assist validation of producers by providing interpretations of ANDF code. The final goal of the GAI is to run HLL test-suites for a specific language on top of it to validate producers.

It is worth noting that the AVS and the GAI could be used to validate other possible ANDF tools such as an ANDF-to-ANDF optimizer, etc.

7. ANDF Validation Suite

This section presents the ANDF Validation Suite (AVS), a collection of hand-written test cases to check installer conformance to the ANDF specification.

Installer validation is an important aspect of ANDF validation:

- It is essential that installers strictly conform to ANDF semantics, as no tuning is possible on the target platform: once one gets a shrink-wrapped ANDF application, it must be installed by plug and play.
- It is the only tool actually producing code that can be run and consequently is the only candidate for testing.

AVS overview

The constructs have been divided into classes, and each class is tested separately:

- namespace constructs;
- token expansion;
- conditional expansion;
- flow control;
- memory accesses;
- pointer arithmetic;
- integer arithmetic;
- floating-point arithmetic.

We have written the tests directly in ANDF, through an ASCII to binary tool which we developed separately. Thus we are independent of any HLL. Most of the tests are hand-written test cases, with the exception of integer and floating-point arithmetic tests, which were generated semi-automatically.

We have a minimum API requirement for the tests themselves. We need some basic I/O for the tests, at least to print out the results. To do so, our ANDF code uses a very limited subset of the ISO-C API [8]:

- Each set of tests is exported as a function named `main()`.
- We need to know the actual definition of the two C types `int` and `char`.
- We output the results through calls to the ANSI-C library function `printf()`.

Coverage

Each construct is systematically tested within its domain of validity, as well as at the limits of its domain. Limit tests include tests such as large number of parameters for a token or a procedure.

Some constructs are strongly related to each other, *e.g.* `div` and `rem`. Such sets of constructs have also been tested altogether.

For arithmetic constructs, each test is repeated three times to stress different optimization schemes of an installer:

Plain constants. Installers are expected to optimize-out many of these tests by constant folding and just generate code that will report a success.

Initialized variables. Clever installers can replace these variables with their actual value by constant propagation and therefore can optimize-out these computations as well.

Dynamic values. The values are passed through calls to a function that just returns their single parameter. In this case, the installer is forced to generate code and the test is actually performed at run-time.

Other tests for installers

There are some test areas that are not yet covered by the AVS:

Complex optimizations, such as optimization of loops.

Typical HLL patterns. Such as structure layout.

We feel that these cases are already covered by test suites specific to HLL, and that for the time being, it is preferable to concentrate on other parts of the AVS.

Outcome

The process of writing the AVS was not without some difficulties. Some were just minor points, others were more critical:

- While writing tests, one has to forget that one is using an actual installer on one's development machine: "*I must forget that I am using a 32 bit RISC machine!*".
- The systematic testing of features yields some bizarre test cases: "*What is the purpose of the test here? Which HLL will need this anyway!?*".
- As we were running our tests on a current version of the technology, we were often one bug ahead of the installer. This means that once a bug is found by a new test, it can be difficult to develop the follow-on test without having anything to run it on as a sanity check.

In the end, writing tests for the AVS turned out to be an interesting process that has raised a number of interesting issues that resulted in clarifications of the wording of the ANDF specification. Needless to say, the quality of installers has also improved during the process.

8. General ANDF Interpreter

Producers are more difficult to test independently of installers. A producer takes high-level language (HLL) source code and generates ANDF code. Note also that this generated ANDF code embeds references to tokens to be expanded on the target platform. Given this, the conformance of a producer means that it generates ANDF code actually reflecting its HLL semantics both by means of pure ANDF code and by the correct use of its HLL Target Specific tokens. Thus, validating a producer per se is not an easy process.

Compilers of a given language are usually validated through a specific validation suite. This process validates the compiler running within its run-time environment. In the case of ANDF, the actual run-time environment is unknown: these environments include any new architectures not yet designed. The point is, regardless of how many environments we can gather today, we will end up with a set of target-dependent environments, which is not the same as being target-independent. This strategy could be an effective pragmatic starting point, assuming that certified installers are available.

To tackle this issue a General ANDF Interpreter (GAI) has been developed. The GAI interprets ANDF code in an architecture neutral way in order to highlight any unspecified platform specific behavior of the ANDF code. It relies on an abstract execution environment to provide arithmetic operation emulation and memory abstraction.

Advantages of the GAI compared with an installer

An ANDF interpreter has many advantages compared to a particular installer on a platform. Its main advantages are that it is simpler than an installer and easier to adapt to specification changes, for the following reasons:

- The semantics of each ANDF construct are defined in a high level language, rather than being expressed by a sequence of machine instructions to be generated by an installer.
- An interpreter does not need to bother with particular hardware conventions, such as calling conventions or memory allocation. Instead, it relies on well defined interfaces and data type abstractions to provide a straightforward implementation of the semantics.
- The semantics of each ANDF construct can be implemented independently of any others, while for an installer the same construct may produce different code depending on the context in which the construct is used. No optimizations -such as register allocation, local and global optimizations, *etc.*- are performed in an interpreter.

Another advantage is that the ANDF interpreter can be parameterized to emulate parts of different architectures. Then, it can be used to check the execution of an application on a particular architecture, or even on non-existent platforms, with unusual word lengths for example. In addition, it can provide a different interpretation for undefined behavior in the ANDF specification, providing “perverse” but legal interpretations.

The interpreter, as it emulates an execution environment, can perform intensive checking, while an installer in most case is limited to the hardware error detection mechanism, which is often not as restrictive as one would like it to be. For example, uninitialized memory cells are flagged, and arithmetic operations are implemented such that they can detect undefined results. Memory accesses are always checked, when reading or writing. Every attempt to access an object beyond its allocation bounds is detected. Also, when calling an API function, which is not interpreted, arguments are checked so that illegal calls can be flagged in most cases.

Problems to be solved in an ANDF interpreter

An interpreter must not only provide the interpretation of the semantics for the ANDF constructs, as an installer generates code to express this semantics at run-time, but also it has to provide an execution environment for the interpretation of the application. There are four main components which the ANDF interpreter provides for the interpretation of an application:

Arithmetic operation emulation, so that the hardware architecture of the platform on which the interpreter is run is not used.This emulation must also be parameterized in order to be able to emulate various execution environments. However, this has not yet been totally achieved for the floating-point arithmetic operations.

An interleaving mechanism, to reflect the fact that the ANDF language is rather liberal in permitting reordering and indeed interleaving of expression evaluation. The GAI supports interleaving of expressions where it is allowed by the ANDF specification. This is used to detect code with dependence on evaluation order. Interleaving has only recently been added in the interpreter. It still needs some development mainly to allow the user to detect and resolve any conflicts between interleaved branches.

A memory abstraction, in order to be independent of the data types of the hardware platform and from their alignments or storage constraints. This can be used to implement unusual data representations or alignment rules.

An API implementation, for the application to be interpreted. The GAI uses directly the API of the platform. Another solution would have been to provide an interpretation of the API functions, either by obtaining an ANDF version of them, or by directly recognizing them in the interpreter and emulating their actions.

Difficulties

Using the GAI in conjunction with HLL test suites to validate producers is more complicated than using the AVS to validate installers.

In the case of the AVS, we have three entities interacting, the AVS, the installer and the hardware. Hardware problems can usually be excluded, the few lines of the AVS involved are easily inspected, so that the attention can be quickly focussed on the installer.

In the case of the GAI, we have three software entities involved: a few lines from the HLL test suite, a producer, and the GAI. Just as for the AVS, the few lines of the test suite can be easily inspected. The problem is that the producer and the GAI are tools of similar complexity, so it is difficult to decide which of them is at fault.

9. The AVS and the GAI

The AVS is used to validate installers. It has to check that every ANDF construct is actually implemented in conformance with the specifications.

The GAI is designed to validate producers, and also any tool which makes ANDF to ANDF transformations. It must check that the semantics of the code generated by a producer expresses the application semantics without any particular architecture assumptions.

However, in order to fulfil their role, these two tools have also to be validated. The AVS may contain code which is not independent of the platform on which it is run. The GAI may fail to interpret some unusual patterns of ANDF. The validation of these tools could be achieved by checking them against many producers and installers to exercise them on a large set of configurations.

But a much better solution is to validate them together. Indeed, as the AVS should cover the entire ANDF specification, with usual and unusual ANDF patterns, it will exercise the whole interpretation code of the GAI. And as the GAI should be able to emulate a large set of environments, as well as non-existent ones, it will check that the AVS does not make any assumptions about the platform it is executed on.

We carried out this validation process, and it allowed us to correct a number of bugs which were not detected by the installers on which we ran the AVS. As we did not yet validate the interpreter on benchmarks or validation suites, the AVS was very useful for detecting problems in the GAI. This work is still in progress.

10. Related work

This work took place within the CEC sponsored OMI/GLUE Esprit project, more precisely in a workpackage dedicated to ANDF specification and validation.

The starting point of this workpackage was the ANDF specification provided by DRA in plain English [4], also called the informal specification. Besides the AVS developed by OSF and the GAI developed in collaboration by Etnoteam and OSF, another important achievement of this workpackage is the Formal Specification of ANDF developed by DDC-I [5]. DDC-I used Action Semantics [6] and the RAISE Specification Language formalism [7] as modeling tools.

Other teams have been developing ANDF components, either as part of the GLUE project or independently. The various discussions resulting from having different teams working on the implementation of ANDF-related tools resulted in increasing quality of, both formal and informal versions of the ANDF specification.

Also, the various discussions with DDC-I about the Formal Specification have been very fruitful, and our modeling of interleaving in the GAI owes much to their work.

ANDF components currently being developed include:

- Installers: MC680x0, 80x86, mips, Alpha, PowerPC, HP/PA, ARM.
- Producers: C, Fortran 77, Ada, Dylan, C++.
- An investigation to re-use existing compiler technology to build ANDF installers: GANDF [2], based on `gcc` from the Free Software Foundation.

11. Conclusion and Further work

Having the AVS and the GAI exercise each other will raise the level of confidence that they both implement the same semantics. But there is no way to ensure that this semantics is the exact ANDF semantics. However, the dynamic semantic description of the ANDF constructs appears to be rather close to the action performed in the interpreter to interpret the ANDF constructs. This could be used to check manually that the GAI implements the right semantics. In many cases, it may even be possible to automatically derive the interpreter code from the ANDF Formal Specification, but this has not been studied yet.

12. Bibliography

A number of papers on ANDF are available from the OSF-RI WWW server:
"`http://riwww.osf.org:8001/index.html`".

[1] *ANDF Validation Suites Specification*, Frédéric Broustaut, Christian Fabre, François de Ferrière, Éric Ivanov, OSF Research Institute, Grenoble, March 1993.

[2] *GANDF: Status and Design*, Richard, L. Ford, OSF Research Institute, April 1993. Available through OSF-RI's WWW server.

[3] *The Ada compiler validation capability.* John Goodenough, Softech, Inc., December 1986.

[4] *TDF Specification (Issue 2.1, June 1993).* Defence Research Agency, Malvern, U.K.

[5] *Formal Specification of ANDF, existing subset*, document code 202104/RPT/19 issue 2, January 1994. Jens Ulrik Toft & Jens P. Nielsen, DDC-International A/S, Gl. Lundtoftevej 1B, 2800 Lyngby, Denmark.

[6] *Action Semantics*. Peter D. Mosses. Published as n°26 under the *Cambridge Tracts in Theoretical Computer Science* by Cambridge University Press. 1992. ISBN 0-521-40347-2.

[7] *The RAISE specification language*, published in *The BCS Practitioner Series* by Prentice-Hall. 1992. ISBN 0-13-752833-7.

[8] *Programming language - C*, International Standard ISO/IEC 9899: 1990 (E), First edition 1990-12-15.

[9] *Validation and Verification program for ANDF.* Frédéric Broustaut, Christian Fabre, François de Ferrière & Eric Ivanov. OSF's ANDF collected papers. October 1993. Available through OSF-RI's WWW server.

[10] *Porting ANDF to Microsoft Windows NT.* Richard L. Ford & E. Andrew Johnson. OSF's ANDF Collected papers. December 1993. Available through OSF-RI's WWW server.