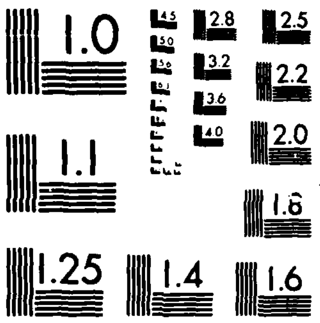
AD-A163 880 EXTENDING DATA TYPING BEYOND THE BOUNDS OF PROGRAMMING LANGUAGES(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) M STANLEY SEP 85 RSRE-MEMO-3878 1/1

UNCLASSIFIED DRIC-BR-97889 F/G 9/2 NL

END

FILMED

DTIC


1.0
4.5
5.0
5.6
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

UNLIMITED

BR97889

2

RSRE

MEMORANDUM No. 3878

ROYAL SIGNALS & RADAR ESTABLISHMENT

EXTENDING DATA TYPING BEYOND THE BOUNDS OF PROGRAMMING LANGUAGES

Author: M Stanley

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3878

TITLE: EXTENDING DATA TYPING BEYOND THE BOUNDS OF PROGRAMMING LANGUAGES

AUTHOR: M Stanley

DATE: September 1985

SUMMARY [illegible] 1)

This paper discusses the use of strongly typed values at the operating system and command language level in a Programming Support Environment (PSE). It describes the use of flexible data structures on filestore, with support for data typing from the operating system and the command language interpreter, as implemented in the Flex PSE developed at RSRE, Malvern. (Great Bri

A

TITLE: Extending data typing beyond the bounds of programming languages.

CONTENTS.

TITLE: Extending data typing beyond the bounds of programming languages.

1. Introduction

Strong data typing is now accepted as a feature of high level programming languages. Programmers have the ability to define a variety of data types for use within a program, to suit a particular problem and appropriate use of values of each type is enforced. Similar facilities for defining and using typed values at the operating system and filestore level would be useful. One of the limitations of conventional operating systems is the inability of such systems to support flexible data typing when the data is to be passed between programs, stored on filestore or processed by the command interpreter. The Flex programming support environment (PSE) developed at RSRE, Malvern, supports the definition of and use of data types (modes) for filestore objects and appropriate use of modes is supported by the Flex command interpreter. The variety of data structures that can be explicitly defined is unbounded. This paper discusses the Flex mode system and indicates how it supports the programmer in his task. [illegible]

([illegible])

2. The need for data structures

In order to pass data safely from one program to another, each program must correctly understand the structure of the interface data. If this structure is hidden within an unstructured data file it is impossible to be certain that the received structure is as expected. Most operating systems, unable explicitly to define or to handle a wide variety of structures, cannot check the validity of interfacing data structures. A user who mistakenly gives a parameter of unexpected form to a tool or program may find that the program misinterprets the data with unanticipated or undefined results.

We need a user interface that protects a user (as far as possible) from the consequences of his own mistakes. A PSE therefore needs facilities for creating data structures and for passing structured data easily from one program or tool to another. It should be possible to use structured data with any tool or program in the PSE including in the user interface. The operating system and user interface should check that values are used in accordance with their defined data type.

Data structures on most filestores are limited to a small number of predefined file types (for example source, binary, loadable program and data file). Files (i.e. objects in filestore) may have their own internal structure, but the structure is invisible to the command interpreter and to the user. It is checked only within individual tools or user programs. Similarly, most command interpreters can handle only system-defined data structures. Few operating systems support the introduction of user-defined data types.

Most operating systems organise the filestore in fixed size blocks (e.g. 512 byte blocks). The mapping of small data structures onto fixed size blocks can lead to very inefficient use of space. This inhibits the introduction of a wide variety of small data structures.

3. What is Flex?

Flex is a multi-language Programming Support Environment (PSE) with a large amount of software available to users. It is built on the Flex capability object oriented architecture developed at RSRE, Malvern. The main design aim was to simplify the development and maintenance of complex software, with a high regard for system integrity and reliability. The result was a highly interactive PSE that is noticeably different from other PSEs. The PSE development since the first Flex architecture came into use in 1978 has been mainly a response to requests from programmers using the system. The software base includes all normal operating system facilities and many other procedures including compilers for Algol68 and Pascal. An Ada(*) compiler is near completion and an ML compiler is under development. A Flex PSE is also referred to as a Flex system.

The Flex capability computer architecture [1,2] has (so far) been implemented in microcode on four hardware configurations, the most recent being the ICL Perq. The implementation with which I am most familiar is a multi-user system in which 3 Flex computers share a common filestore and common peripherals.

In Flex there is no distinction between tools, programs, utilities and procedures. Consequently this paper will normally refer to utilities and to other tools as procedures. Most of the operating system procedures, such as the editor and the command language interpreter (curt) [3] can be called from user programs as well as from the command language interpreter.

A full description of Flex is beyond the scope of this paper, which will concentrate on those aspects that contribute to data typing at the operating system and filestore level, with some discussion of the effect this has on programmers and on the method of use of the Flex PSE.

* Ada is a registered trademark of the US DoD.

4. Modes on Flex

4.1 Modes and values

Built on top of the Flex architecture, as a normal unprivileged program, is an object oriented operating system. Every object or value handled on Flex, whether on filestore or in mainstore, has an associated Flex mode (value type) that is used to indicate how the value is to be interpreted, and the operations that are valid on it. Values vary in type from simple integers to much more complicated structures . Atomic modes are used to describe values whose structure need not be visible to the user.

In contrast to the limited set of filestore types available in most computer systems the variety of Flex modes is boundless. Flex architecture supports the use of values of any size whether in filestore or in mainstore. Values do not need to be fitted into fixed size blocks as in so many operating systems. Most mainstore objects (including procedures) have direct analogues in Flex filestore. The ability to hold on filestore values of such a wide variety of Flex modes gives much greater flexibility than encountered with conventional filestore.

The basic modes provided by Flex include analogues of all the modes and data types encountered in modern programming languages as well as some modes peculiar to Flex. The user can also invent additional modes (that are composed of the basic modes) to suit his problem. The additional modes may be atomic or they may have explicit visible structure according to user requirements.

For example, some of the basic modes are:

Int (the value is an integer);
Char (the value is a character).

There are also mode constructors, to permit more complex modes to be defined. For example:

(Int,Real,Bool)	(the value is a structure containing an integer, a real number and a boolean value);
Vec Int	(the value is an ordered set (a vector) of integers);
Vec Vec X	(the value is a vector of vectors of values of some mode X).

In addition there are modes, peculiar to Flex, some of which are discussed below.

4.2 Capability modes

There is a special kind of value on Flex called a capability. When a user wishes to use any other kind of value on Flex he needs a capability for that value. Capabilities are fundamentally different from all other values in that they can be created and modified only by the Flex microcode and are used to control access to Flex values and to prevent inappropriate use of them. There are mainstore capabilities that control access to mainstore objects, filestore capabilities that control access to objects on filestore and remote capabilities that control access to remote facilities (e.g. objects on other computers).

In a sense a capability is a pointer created on behalf of the user by the microcode, but the capability also contains information on the type of use (read only; read/write; execute) that will be permitted by the microcode.

The mode of a capability value indicates the mode of the value to which it gives access. For example

Edfile (the value is a capability to read a file (an edfile) that can be handled by the editor);
Module (the value is a capability for an object (a module) that gives access to compiled code and to the source text from which it was derived);
Vec Edfile (the value is a vector of capabilities for editable files);
Ptr X (the value is a mainstore capability for a value of mode X).

The Flex architecture ensures that the only software or user operations involving a capability are:

1. request a new capability from the microcode (any procedure can do this; it is not a privileged operation);
2. store a capability for future use;
3. copy the capability to another user;
4. delete a copy of the capability;
5. use the capability as authorised by the microcode.

4.3 Procedure modes

Procedures (or programs) on Flex are values and are treated like any other value. Every procedure on Flex therefore has an associated Flex mode. The value of a procedure is a capability to execute the procedure and the possession of a procedure capability allows the holder only to execute the procedure. It does not allow him to dismember the procedure to find how it works, what other procedures it might use or the values of its non-locals. This is the basis of much of the security of data in Flex filestore and mainstore.

The Flex mode of a procedure is a triple written as:

input_parameter_mode -> result_mode

where the symbol -> is one of the elements of the triple and shows that the value is a procedure.

There is an empty value on Flex whose mode is Void. A procedure on Flex that apparently delivers no result actually delivers the empty value. The Flex mode of such a procedure is written:

input_parameter_mode -> Void

For example, a procedure to list an edfile on a printer would be of Flex mode: Edfile -> Void
and a procedure that takes no input parameters has Flex mode:

Void -> result_mode

A procedure to count the entries in a vector of integers would be of Flex mode: Vec Int -> Int

Although a procedure cannot itself be a filestore object there is an analogous filestore object, called a filed procedure, of Flex mode:

Filed(procedure mode)

Before calling a procedure or program, curt checks the Flex mode of the input parameters. Any attempt to apply a procedure to a value of the wrong Flex mode fails, so users can have confidence that procedures will not be applied to unexpected data structures.

4.4 Using a Flex mode as a value

A mode can also be regarded as a value (of Flex mode Mode) that can be passed into and out of a procedure.

Sometimes a procedure may need to examine the Flex mode of its input. Consider a procedure intended to handle correctly a value which may be of any mode. A special mode, called Moded is available in which any value, V, of any mode, M can be expressed as a moded value. The representation of the Moded value contains both V and M. Thus the mode of a value can be passed into and out of procedures together with the value. Curt can, when appropriate, convert a value V of mode M into a Moded and it can untangle a Moded to deliver the value V in mode M.

For example, consider a procedure "show" that takes any Flex value and displays it on the screen in a form appropriate to its mode. The Flex mode of "show" is: Moded -> Void. The procedure "show" untangles the Moded (V, M) and displays V in appropriate form. If "show" is called from curt the input may be of any mode, because curt will convert the value to Moded form.

Consider a procedure "find" which takes a name and searches a dictionary for a value associated with the name. The value may be of any mode. The procedure delivers the value together with its mode, as a Moded value. The Flex mode of "find" is: Vec Char -> Moded. Curt will untangle the Moded and deliver the value in the correct mode.

One result of embedding the mode information with the value in the Moded mode is that a single procedure can handle a boundless variety of data structures instead of needing a different procedure for each different structure. This reduces the quantity of software needed to handle different data structures, and also allows existing functions to handle new modes created since the function was written.

4.5 Choosing the Flex mode of an object

The Flex mode of an object is information to the command interpreter, and to other procedures, on how to interpret a value. The utility that creates a procedure can select a Flex mode for the procedure that matches the external interface of the procedure as defined by its source text. However, since Flex modes are richer in scope than the modes available in most programming languages, in particular in the ability to distinguish a capability value from other pointers or integers, a user may prefer to define the procedure mode for himself. For example he may wish to use the procedure mode to protect a procedure that expects a capability from being called with an integer parameter even if the programming language treats the capability as an integer.

A Flex mode can legitimately be changed, and facilities are provided to do this. This enables a user to select modes that accurately reflect the use to which the value is to be put, such as changing a procedure mode to deliver a value of mode Edfile instead of mode Integer. The value is unchanged by a change of mode. All that is changed is the interpretation of the value. Since the mode of a value on Flex is advisory, a user can select any mode whether or not the mode can in fact be used to interpret the value. If a mode is supplied which cannot match the value it will be rejected when curt attempts to interpret the value. System integrity does not rely on correct modes. You cannot dismember a procedure, for example, by giving it a different Flex mode.

4.6 Exceptions

Sometimes a procedure will fail, because of an internal error (such as dividing by zero), an explicit failure or an Ada exception. When this happens, instead of returning a value of the expected type to the calling procedure an exception value will be returned. The calling procedure can handle the exception internally or it can pass the value out through the calling procedures to the command interpreter. When this happens, a value of Flex mode Exception will be returned to the command interpreter.

A procedure diagnose, of mode Exception ->Void can be called from curt or from a user procedure to present to the user information on the state of the procedure at the point of failure.

The ability of Flex to handle exceptions as normal values allows them to be treated in a uniform way by procedures. This reduces the amount of special software needed in an application that caters explicitly for exceptions as well as providing necessary information to the diagnostic procedure.

5. Summary

The concept of Flex modes and the procedures for creating them and handling them give excellent facilities for creating and using data structures to suit the problem. The variety of modes of both filestore values and mainstore values is unbounded. The flexible blocking of filestore means that the use of small structures will not result in an intolerable waste of filestore space. The command interpreter provides checking of the Flex modes similar to the type checking provided by a high level programming language.

New types of data structure are just new Flex modes. Augmenting an environment with new Flex modes has no unwanted side-effects. It does not impose changes on existing procedures and modes. Any data structure can be constructed as a Flex mode. The user can declare a new atomic mode if he wishes to hide the internal structure of values of that mode. The Moded mode, which binds together a value and its mode, so that the user program can interpret the value in accordance with the mode to which it is bound, is a very powerful facility. It allows general purpose procedures to be written, to handle objects of any mode, thus reducing the amount of software needed in total.

The command language (curt) encourages users to pass structured data from one procedure to the next. The compatibility of the Flex mode of the procedure and the Flex mode of the input data are checked by the command language interpreter, reducing the risk of passing an incorrect data structure between procedures. This is a useful protection for careless users. It also reduces the amount of software needed in user programs, which no longer need to check the data structure of the input data.

6. Conclusions

The Flex mode system provides at the operating system and filestore level the data structuring and associated data type checking normally associated with a high level programming language. The ability to create new Flex modes and to handle objects of such a wide variety of modes both from the command language and as filestore objects gives great flexibility. The user can handle data structures not anticipated when the system was developed without losing the benefit of having Flex modes defined for procedures and checked by the command language interpreter. The amount of software needed for data checking in user programs is reduced because the command interpreter will not pass values of incorrect mode to a user program. This gives confidence that programs will not give unexpected results as a consequence of misinterpreting the interface data.

The Flex mode system is sufficient not only to describe all data types in most modern programming languages but also to accomplish the more challenging task of describing values used in the command language. The use of modes for capabilities and of mode constructors for a value contribute to the power of the Flex mode system to cater for the needs of the command language. Moded values that allow any value to be bound to its mode is a particularly powerful extension to the usual concept of data types. The use of Moded values as procedure parameters removes the need to provide several different procedures to perform essentially the same function with values of different mode. It enables a single procedure to use the embedded mode information to interpret the value of an input parameter and it permits a procedure to deliver a values of different modes as Moded values.

The improvement in software productivity resulting from use of a PSE such as Flex, with the power of the mode system to assist the programmer, has not been measured, but the absence of many of the problems that beset programmers on conventional systems must result in saving of effort and improved productivity.

The mode system just described is but one of several unusual and useful features of the Flex PSE. Future work on Flex is aimed at making the PSE and the ideas it demonstrates more widely available, and at improving the facilities. The underlying architecture is not expected to change, but additional facilities are being worked on to enable Flex to be networked and to support host/target software development. The possibility of implementing Flex with its powerful mode system on existing computer systems (without re-microcoding) is being considered as a topic for future research.

7. References

1. "Flex Firmware" by I.F.Currie, P.W.Edwards and J.M Foster. RSRE Report 81009. Sept 81.

2. "Flex: A working computer with an architecture based on procedure values." by I.F.Currie, P.W.Edwards and J.M Foster. RSRE Memorandum 3500. 1982.

3. "Curt: The command interpreter for Flex" by I.F.Currie and J.M.Foster. RSRE Memorandum 3522. 1983.

AD-A163880

DOCUMENT CONTROL SHEET

Overall security classification of sheet .... UNCLASSIFIED ....

(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 Memorandum 3878	3. Agency Reference	4. Report Security Classification U/C
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location ROYAL SIGNALS AND RADAR ESTABLISHMENT		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		
7. Title Extending data typing beyond the bounds of programming languages.			
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 Stanley, M	9(a) Author 2	9(b) Authors 3,4...	10. Date 9.1985	pp. ref.

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

15. Distribution statement

UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This paper discusses the use of strongly typed values at the operating system and command language level in a Programming Support Environment (PSE). It describes the use of flexible data structures on filestore, with support for data typing from the operating system and the command language interpreter, as implemented in the Flex PSE developed at RSRE, Malvern.

S80/48