PUC

Series: Monographs in Computer Science
and Computer Applications
Nº 3/76

ON MONITORING THE USE OF DATA

by

Arndt von Staa

Departamento de Informática

Pontificia Universidade Catolica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

Series: Monographs in Computer Science
and Computer Applications
Nº 3/76

ON MONITORING THE USE OF DATA*

by

Arndt von Staa

Series Editor: Sergio E.R. Carvalho February, 1976

* This research was supported in part by the Conselho Nacional do Desenvolvimento Científico e Tecnológico (CNPq-Brasil) and the National Science Foundation (NSF-USA) interchange program, and developed by the Information Systems Group of the Departamento de Informática - PUC/RJ.

Copies may be requested from:

Rosane T.L. Castilho, Head
Setor de Documentação e Informação
Departamento de Informática - PUC/RJ
R. Marquês de São Vicente, 209 - Gávea
20.000 - Rio de Janeiro - RJ - BRASIL

ABSTRACT:

The concept of monitoring the use of values of a given type is introduced. Operation emulating functions are introduced as one of the ways to permit monitoring. Demons, breakpoints and mousetraps are defined and an implementation of these by means of operation emulating functions is presented. More evidence is gathered showing that hardware support is beneficial to implement these programming tools.

KEYWORDS:

Modularity, Type Descriptors, Monitoring, Operation Emulating Functions, Demons, Breakpoints, Mousetraps.

RESUMO:

Apresenta-se o conceito de monitoramento do uso de valores de um determinado tipo de dados. Como uma das possíveis soluções para a implementação de monitoramento apresentam-se funções emuladoras de operadores. São definidos, também, fantasmas, pontos de interrupção e armadilhas, sendo apresentada uma solução para implementação dessas ferramentas utilizando funções emuladoras de operadores. É enfatizada a utilidade do suporte de "hardware" para concretizar a implementação dessas ferramentas de programação.

PALAVRAS CHAVE:

Modularidade, Descritores de Tipo, Monitoramento, Funções Emuladoras de Operadores, Fantasmas, Pontos de Interrupção, Armadilhas.

CONTENTS

1 - INTRODUCTION

In several cases we may want to control the use of values transmitted between program modules. For instance, we may want to forbid copies from being made, or we may want to establish reference counts. In order to achieve this, we will introduce the concept of operation emulating functions. Such functions create new type descriptors by replacing existing functions of the base type descriptor and allow the effective monitoring of the use of values of this new type.

We will also introduce the concept of demons breakpoints and mousetraps [Rosenberg 75]. We show, then, how these programming tools could be implemented by means of operator emulating functions. By closely examining the difficulties arising from this solution method more evidence is gathered supporting the need for hardware aids for the implementation of these programming tools.

2 - MODULARITY

In order to achieve modular programming, a program module, or simply a module, must possess at least the following properties [Cowan et alii 76]:

a - syntactic non-interference [Dennis 73] - the property which assures that modules may be combined without the necessity of textual changes in any of the combined modules, e.g. changing textual names, within modules or module names themselves.

b - semantic context independence [Dennis 73] - the property which assures that no data space definition has to be made outside the modules other than parameter data spaces.

c - data generality [Dennis 73] - the property which assures that modules may be combined on the basis of abstract data types rather than in terms of their implementations.

d - implementation hiding [Parnas 71] - the property which assures that modules can be fully understood exclusively by means of their interface description (information).

e - low coupling [Myers 73] - the property which assures that all parameters defined within the interface information are effectively used by the module.

f - high strength [Myers 73] - the property which assures that each module is precisely defined and that all functions it performs are strongly interconnected in a logical sense.

Our characterization of a program module is similar to the one introduced by SIMULA with its class [Dahl et alii 72] or by CLU with cluster [Liskov et al 74].

Specifically a program module defines:

a - internal data spaces, internal type descriptors and internal functions which are not visible externally to the module;

b - visible type descriptors and functions (but not data spaces) which are made explicit to the exterior of the module, i.e. are within the outside scope of the module.

Modules may act as type descriptors, or simply as types, e.g. integers, stacks, trees etc. For example, the type integer can be described by a module defining an internal data space - a word - and the visible operations copy, fetch, add etc. An integer declaration is then nothing more than the creation of an instance of the module describing this type, e.g. an object of a SIMULA class.

Type descriptors are information and, consequently, may be considered values of type type. Such parameters allow the practice of modular programming, in which the implementation characteristics of a given type descriptor are effectively hidden from its users. In most cases, type checking will be performed statically. The conditions under which this is possible are examined in [von Staa 74].

```
type: node_manager(type: node_type) is
     begin node_manager;
          type: node is node_type;
          node_type array: area[1::100];
          outside_scope operations;
               type: ref is shield integer;
               ref: function allocate(node_type: data) is ...
               ref: function null is null:=0;
                    function free(ref: whom) is ...
       node_type:   access function deref(ref: who) is ...
          end operations;
     end node_manager;
```

FIGURE 1. Example of a type descriptor module with a parameter of type type.

Figure 1 shows an example of a type descriptor which uses a parameter of type type. Due to this parameter we may implement the module "node_manager" without having to define completely the information stored within each node. Consequently this "node_manager" can be used for several different purposes, possibly by one and the same program or module.

In figure 2 we show an example of how the module "node_manager" of figure 1 could have been used. It should be clear from these two examples that "user_type" needs no information about "queue" and "node_manager". Conversely these descriptors are independent one from the other and also from "user_type". Of course several problems may arrise when attempting to define an array of "user_type" since not necessarily will all node sizes be equal. Although important for an efficient implementation, we will not deal with these problems here since this would be beyond the scope of this report.

```
type: queue(type: user_type) is
     begin queue;
          requires type: node_manager;
          type: node is struct(area.ref: next; user_type: info);
          node_manager(node): area;
          area.ref: head, tail;
          head:= tail:= area.null;
          outside_scope operations;
               function dequeue is ...
       user_type: access function update is
                    begin update;
                         if head=area.null
                            then signal error("accessibility");
                            else update:=info in area.deref(whom);
                         fi;
                    end update;
          end operations;
     end queue;
```

FIGURE 2. Using "node_manager".

---

3 - MONITORING

In several cases we may want to control the uses made of values of a given type. For example, we want to maintain reference counts. In this case all operations which might generate a copy of a reference must be monitored, since, otherwise, the correctness of the reference counts cannot be established. It must be noted here that copies of references might well be made due to hidden copy operations, such as the passing of parameters by value.

The operations used by language processors are identified by some textual name, e.g. '+', ':=', 'log' etc. When monitoring an operation 0, we must be able to replace the original operation implementation by a new implementation 0'

bearing the same textual name. The new operation 0' will be called 0-emulating function. Some current languages allow such replacements, e.g. SNOBOL 4 "OPSYN" function [Griswold et al 71], ALGOL 68 op declaration [van Winjgaarden et al 69]. The textual name may be implied by the language processor. For example, the assignment operator ':=' is usually known globally and activates the function 'copy' defined within the target type.

The language processor used could have been defined by the user by means of another more primitive language processor. We have then a hierarchy of language processors. It follows from this discussion that the operation to be replaced may well have been user defined at some lower level language processor in the hierarchy.

For each module some descriptive information is available. This information will be called the module interface information. This module interface information defines among other things, the parameters and the types of the parameters required by the program module. It also defines the type and name of the operations made available in the outside scope of the module. When combining modules in a strongly typed environment the types of corresponding parameters must satisfy the type checking conditions. Usually these conditions require equality of the type descriptors. A detailed set of conditions is given in[von Staa 74].Furthermore, conversion functions may be interposed between combined program modules in order to allow the conversion between different implementations of the same abstract data type [von Staa 75].

Let 0 be an operation defined in the type descriptor τ. In order to define an 0-emulating function 0', the implementation of τ must be known. Hence 0' is either defined within τ or is defined as an extension of τ yielding a new type τ', in which 0 has been replaced by 0'. It follows from this discussion that 0-emulating functions are always defined within some type descriptor. Consequently 0-emulating function binding can be performed by the type checking and transmission mechanisms.

With respect to defining 0-emulating functions, two

main difficulties arise immediately:

i - the replacement may cause language processor assured type-wise correctness to be lost.

ii - some operations are such that the emulating function must either report an error or eventually perform the operation being emulated. For example, emulating the deallocation operation must eventually perform this deallocation, otherwise disastrous accesses could occur.

From (ii) it follows that an 0-emulating function 0' may have to refer to the operation 0 itself. On the other hand, 0 and 0' are identified by the same textual name in order to allow the replacement to be performed when matching the interface information. We must then adopt the following convention in order to avoid improper replacements:

Convention: Let TX be the textual name of some operation 0 and also of its 0-emulating function 0'. Within the body of 0', any occurrence of TX refers to 0 and is not replaced by 0'.

An 0-emulating function 0' will be said to be exactly 0-emulating, if when elaborating 0', 0' either reports an error or the operation 0 is performed with exactly the same parameters as those passed to 0'. It should be clear, that an exactly 0-emulating function maintains the type-wise correctness of the language processor. This follows from the fact that the original operation 0 is necessarily type-wise correct, otherwise the language processor would be in error. Furthermore, this operations is eventually performed whenever 0' completes elaboration without reporting an error. Thus, exactly 0-emulating functions cannot "invent" values which are non-existent in the original type.

Many 0-emulating functions are quite simple. Thus, if some textual discipline is enforced, these functions could be shown to be exact in a mechanical way, without increasing

the compiling cost by much. In order to avoid misuse of the programming language, we may require that some protection convention be satisfied whenever defining an 0-emulating function which cannot be mechanically proved exact. In doing so we establish a hierarchy of programmers and of responsibilities.

```
extend node_manager with
     begin extension;
          replace type: node is struct(integer: reference_count;
                                       node_type: value);
          emulate ref: target := ref: object is
               begin assignment;
                    ref: save_pointer := target; /*save original value*/
                    target := object;          /* perform assignment*/
                    if target≠null
                      then reference_count in @target+:=1;
                    fi;
                    if save_pointer≠null
                       then reference_count in @save_pointer-:=1;
                    fi;
                    if reference_count in @save_pointer=0
                       then (value in @save_pointer).
                                       nullify_pointers;
                            /* link node into free list */
                    fi;
               end assignment;
          emulate ref: allocate ...
     end extension;
```

FIGURE 3. Extending "node_manager" to perform reference counts

In figure 3 we show how reference counts could be maintained by means of defining an assignment (':=') emulating

function. The operation to be emulated is the integer assignment operation which is implicitly defined within the type "ref" of "node_manager". By virtue of this extension, the type descriptor "node_manager" becomes a new type descriptor possessing a different identification, although the textual name might be the same. The maintenance of reference counts is made possible due to the replacement of the type "node" in the original "node_manager" by a new type "node" which possesses an explicit field for reference counts.

When using reference counts, a data space is deallocated whenever its corresponding reference count I reaches the value zero. It follows immediately that the reference count J of the data space β must be decreased if α contains a reference to β. Now, for reasons of data generality, we do not want to know the "node_type" of the data space α being deallocated. It follows then, that "node_type" must provide the means to set the references (data space identifications) contained within α to "null". The easiest way is to define within "node_type" a function "nullify_pointers" which stores an adequate "null" value in all name typed fields of the data space being deallocated. These store (':=') operations will then be monitored again. Thus, by means of recursion, eventually all data spaces to be deallocated will be effectively deallocated. Observe that, by means of "nullify_pointers", a given data space could provoke the updating of several "dynamic storages". It follows then, that O-emulating functions must be implemented in such a way that they allow recursion, i.e. multiple instances. This could be achieved, for example, by means of "safe routines" as in IPL V [Newell et al 65].

Notice that "node_type" may stand for a hierarchy of types. Thus, "nullify_pointers" might be partioned into several sub-operations, emphasizing that "nullify_pointers" be a language processor defined operation.

Notice that "null" is in fact a function which is defined for data space identification types, e.g. pointers, references, indices, etc. It follows then, that "nullify_pointers"

needs only to modify those fields of the data space α being deallocated which currently bear a type defining the operation "null". Thus, fields containing only computational values need not be changed, reducing thus the deallocation run-time cost.

4 - TRANSMISSION OF MONITORS

Following the terminology described in [Rosenberg 75] we will define and examine several programming tools which are based on monitoring.

A demon is a function (exception handler) which is activated whenever given data spaces satisfy specific relations. A brakpoint is a function (exception handler) which is activated whenever a given data space is accessed (read, write or execute). A mousetrap is a function (exception handler) which is activated whenever a value of a given set of values (range) is stored into some given data space.

Conceptually there are no differences between these three software tools, since all three are functions which are triggered when certain more or less complex conditions are met, the reason for the differentiation being the class of conditions which activate each kind of function. We may thus use the term phantom collectively for the three tools mentioned above.

In conventional programming processors, phantoms may cause significant overhead, mainly when the triggering conditions are to be transmitted over module boundaries. Phantoms can be implemented by means of store and fetch emulating functions. We will use the following construct for this purpose:

when <conditional expression> do <phantom body> od;

For this construct to work, all variables occuring within <conditional expression> must be monitored. Consequently the types of these variables must be modified (extended) so as to achieve monitoring capabilities. As mentioned earlier such extensions create new type descriptors. This may result in

type inequality when combining modules, since the type of the actual parameter could be an extension and hence be different from the type of the formal parameter.

Let us examine then how we could achieve the transmission of monitoring types. Let τ_o be some type and let τ_e be a monitoring extension of τ_o, where the only modifications to τ_o are the replacements of type descriptors and of operations by emulating functions. The interface information necessary to perform the binding has not been changed, since no new name has been created nor has a name known through τ_e been deleted. The axioms governing τ_e and τ_o may be different though. For example τ_e could emulate the store operation of the type integer in such a way that only prime integers are stored. Intuitively there is a conversion function from τ_e to integer since the set of prime integers is in fact a subset of the set of integers. This is not necessarily the case in general. For example, τ_e could have chosen to store encodings of the prime integer numbers, e.g. the ordinal number of occurence of the prime numbers. In this encoding there may be prime integers which are too big to be represented by the original implementation of integer. Notice that in this case τ_e is not an exact emulation of the integer store operation since the parameters given to store a prime in this encoding cannot possibly be exactly the same as those presented to the replaced integer store operation.

The existence of the conversions $C_{eo}: \tau_e \rightarrow \tau_o$ and $C_{oe}: \tau_o \rightarrow \tau_e$ [von Staa 75] does also not help. This follows from the fact that if the combined modules communicate via these conversion functions we effectively inhibit the transmission of the signalling <conditional expressions> necessary to implement phantoms.

It follows from the discussion above that type descriptor extensions may have to be effectively transmited accross module boundaries. Consequently such extended type descriptors must be viewed as replacements of the original type descriptors, where these replacements act also over module boundaries. It should be clear now that the when construct shown earlier allows automatic discovery of the variables and the types which are affected by the when contruct, and how these the type descriptors are affected.

A consequence of the above discussion is that compilers no longer may produce object code immediately, since it is not necessarily known how the operations will be implemented at run-time. Furthermore, the use of function calls instead of macro expansions may result in grossly expensive object programs. There is then a need to define a good intermediate language which provides the flexibility required and also allows for code optimization when modules are combined.

Although emulating functions can implement phantoms, there are several drawbacks:

a - it is difficult to insert or remove the <conditional expression> at run-time. Consequently the use of phantoms as interactive programming aids (debugging, artificial intelligence) becomes impractical. In [Rosenberg 75] a proposal is made to trigger the <conditional expression> by hardware. Such a hardware facility removes the necessity to define type descriptor replacements, since now the hardware is used as a linking mechanism between monitored variables and <conditional expressions>. As stated in [Rosenberg 75] the hardware mechanism increases significantly the efficiency with which monitoring may be performed and also permits dynamic changes to the set of variables to be monitored.

b - modules may be combined with several other modules within one single application. This may cause the need for run-time type checking when type descriptor replacements are present. This follows from the fact that the new type descriptor is different from the replaced one, although their textual names could be the same. When hardware is used to trigger <conditional expression> this run-time type checking is not necessary, since the monitoring becomes part of the monitored variable itself rather than being simulated by the type descriptor of that variable. Observe that, when simulating monitoring by type descriptors, we are introducing a subtle syntactic dependence requiring type replacements rather than

simple binding via conversion functions.

5 - CONCLUSION

We have introduced the concept of operator emulating functions and have shown how such functions may be used to establish software monitoring facilities. We have also shown how operator emulating functions could act as exception condition signaling routines. This use of emulator functions imposes inherent difficulties when done exclusively by software means, these difficulties being restriction of flexibility, decrease of run-time efficiency and possible run-time type checking requirements. Furthermore, the exclusive use of software means introduces a subtle form of syntactic dependence due to the need to replace type descriptor rather than connecting modules simply via conversion functions. We concluded from the difficulties mentioned that the implementation of such exception condition signalling routines should receive hardware support.

REFERENCES:

COWAN, D.D.; LUCENA, C.J.; VON STAA, A. On the concept of modules in programming systems. Waterloo, University of Waterloo, Computer Science Department, 1976. CS-76-05.

DAHL, O.J.; MYRHAUG, B.; NYGAARD, K. Simula 67 common base language. Oslo, Norway, Norwegian Computing Centre, 1972. S-22.

DENNIS, J.B. Modularity In: BAUER, F.L., ed. Advanced course on software engineering. Berlin, Springer Verlag, 1973., p. 128-82.

GRISWOLD, R.E.; POAGE, J.F.; POLONSKY, I.P. The SNOBOL 4 programming language, Englewood Cliffs, N.J., Prentice Hall, 1971.

LISKOV, B. et alii. CLU design notes. Cambridge, Mass., Massachusetts Institute of Technology, Project Mac, 1974.

MYERS, G.J. Characteristics of composite design. Datamation, 19 (9): 100-2, Sept. 1973.

NEWELL, A. et alii. Information processing language - V manual. Englewood Cliffs, N.J., Prentice Hall, 1965.

PARNAS, D.F. Information distribution aspects of design methodology. In: INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING, IFIP congress, Ljubljana, 1971, Computer Software Booklet TA-3, Amsterdam, North Holland, 1971.

ROSENBERG, P.A. On demons, mousetraps and breakpoints. Los Angeles, UCLA, Computer Science Department, 1975 (internal report).

VON STAA, A. Data transmission and modularity aspects of programming languages. Waterloo, University of Waterloo, Computer Science Deaprtment, 1974. CS-74-17.

VON STAA, A.; LUCENA, C.J. On the implementation of data generality. Rio de Janeiro, PUC, Departamento de Informática 1975. MCS-75-5.

VAN WINJGAARDEN, A. et alii. Report on the algorithmic language Algol 68. Numerische Mathematik, 14: 79-218, 1969.