# ARASH: A RE-STRUCTURING ENVIRONMENT FOR BUILDING SOFTWARE SYSTEMS FROM REUSABLE COMPONENTS

Abbas Birjandi

T. G. Lewis

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

(503) 754-3273

TR 86-10-2

# Arash:A Re-Structuring Environment For Building Software Systems From Reusable Components

Abbas Birjandi

T.G. Lewis

Department of Computer Science

Oregon State University

Corvallis, Oregon 97331

(503) 754-3273

## Abstract:

Arash is a rule-based tool for re-structuring source programs in order to build software systems from reusable components. Arash incorporates a collection of Generalizers which transform source code modules into abstracted modules. Conversely, a collection of Refiners produce a concrete instance from an abstracted source module. Both Generalizers and Refiners operate on source code components called *fragments* to restructure existing programs, documentation, and associated text. The main significance of Arash is that it is a rule-based tool which can be applied to a family of programming languages, e.g. Pascal, Modula-2, and C. In this paper we describe how to use Arash to restructure Modula-2 source code modules taken from a programmer's database of reusable components in order to construct new software systems, quickly and correctly.

**Keywords:** Programming environment, program transformation, source code mutation, syntax directed tools, code fragments, code selection, customizing, general software, generic systems, program generation, tailoring, reuse of software.

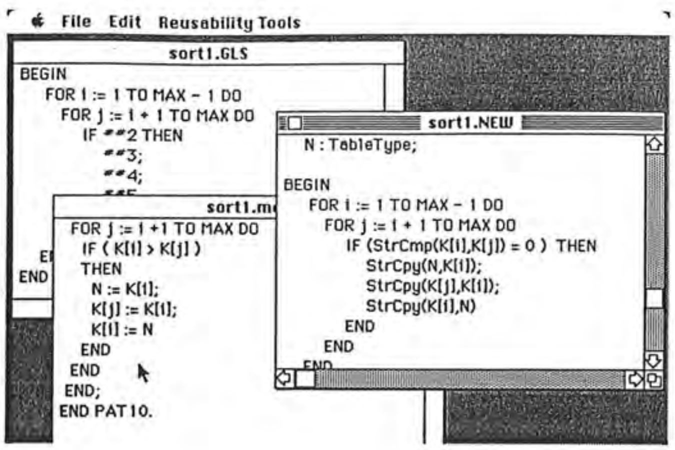
## 1 Introduction

One approach for dealing with the rising cost of software development is to reuse existing software components. The benefits of reusing existing software are: 1) reduction in the cost and development time to produce a new program or system of programs, and 2) an increase in the ease of maintenance and enhancement of existing software systems [Che83].

Arash is an instance of a meta-tool called Yashar described in [BL86] which speeds the development of programs through reuse of existing software components. The basic idea is to construct new program modules from old ones by applying two automatic transformations:*Generalization* and *Refinement.* Generalizers transform a source code module to an abstracted form. Refiners operate on the abstracted form of a module to produce a concrete instance. Figure 1 shows a Modula-2 source program (**sort1.mod**) for sorting an array of integers, its abstracted form produced automatically by a series of Generalizers (**sort1.GLS**), and a concrete instance to sort an array of strings (**sort1.NEW**) generated by a series of Refiners. Notice in the abstracted version the actual abstracted program fragments are replaced by special identifiers called *meta identifiers* (e.g. ##1, ##2,...).

Figure 1: Sort Before and After Generalization and Refinement

Although other source languages might be used with Arash, Modula-2 [Wir83] is used as the source language. In Modula-2 , a module has two parts: 1) a definition part which defines the visibility of constants, types, variables, and procedures of the module which can be accessed by other modules and, 2) an implementation part that encapsulates the actual implementation detail of the module. In addition, Arash supports the concept of an extended module—one that has attributes *.GLS* contains



```
BEGIN
   FOR i := 1 TO MAX - 1 DO
     FOR j := i + 1 TO MAX DO
        IF **2 THEN
           **3;
           **4;
```

```
FOR j := i +1 TO MAX DO
  IF ( K[i] > K[j] )
  THEN
    N := K[i];
    K[j] := K[i];
    K[i] := N
  END
 END
END;
END PAT10.
```

```
   N : TableType;

BEGIN
   FOR i := 1 TO MAX - 1 DO
     FOR j := i + 1 TO MAX DO
        IF (StrCmp(K[i],K[j]) = 0 ) THEN
           StrCpy(N,K[i]);
           StrCpy(K[j],K[i]);
           StrCpy(K[i],N)
        END
     END
```

Fig 1

a reusable module with meta identifiers replacing fragments, *.DRI* contains a list of meta identifiers and the actual code fragments they replaced, *.MLS* contains the rules that are used for the process of generalization, and *.NEW* the newly generated concrete module.

Two other components of this environment are a *Programmer's Data Base* (Grab-Bag) used to search for suitable source programs, and a set of *Browsers* to aid in reading and understanding the existing source programs. The Programmer's Data Base and Browser facilities are the subject of another paper and will not be discussed in this paper [Bir86]. These tools are implemented in C on the Macintosh personal computer.

## Reusability

In [Ker83] reusability is defined as anyway in which previously written software can be used for a new purpose or to avoid writing new software. This definition covers representation of software at both object code and source code level. However, reuse of source code in contrast to object code has the advantage of 1) adapting the interface as well as implementation part of a module to a new interface specification, 2) providing an opportunity to tune, optimize, and eliminate unnecessary code, and 3) providing readable code so that a programmer's knowledge of the reusable module is increased. For these reasons, Arash operates directly on reusable source code components.

This allows the possibility of:

1. Source code reuse/replication by reuse of part or all of existing source code or its data structure,

2. Detailed algorithm reuse by reuse of source code from existing programs as an example of how to do a new program,

3. Large-scale structural reuse by selecting and adapting program design,

4. Maintainability/enhanceability by increasing the effectiveness of programmers by enabling them to study programs with the aid of understandability tools,

5. Portability by facilitating the reuse of software across a wide range of hosts, and

6. Optimization by enabling tuning of generated source code.

## Reusability Life Cycle and Arash

When reusable components are used to build a new software system, the traditional software life cycle is altered. Table 1 shows the difference between traditional software life cycle and reusability life cycle. The additional phases in the reusability life cycle indicate how a designer uses existing components rather than implement everything from the beginning.

Tools in Arash are only applicable for reusing and maintenance of existing source programs. Maintenance may be considered as reusing the original product [Fre83]. In maintenance, problem specification is usually better defined and the reusable module does not have to be found [Fre83]. Problem definition is the phase during which the problem to be solved is formalized as a set of needs; requirement analysis is the process of studying user needs to arrive at a definition of system software requirements; system design specification is the period of time during which the designs for architecture, software components, interfaces, and data are created,

| Traditional Life Cycle | Reusable Life Cycle | Arash Support |
|---|---|---|
| Problem Definition | Problem Definition | None |
| Requirement Analysis | Requirement Analysis | None |
| System Design Specification | Find and reuse similar System Design Specification | None |
| Detailed Design Specification | Find and reuse similar Detailed Design | GrabBag,Browsers |
| Implementation | Find and reuse existing routines from object code library | None |
| | Find and reuse (modified) source code from previous systems | GrabBag,Browsers Generalizers,Refiners |
| | Produce Glue Code | None |
| Testing | Testing | Some help by Browsers |
| System Integration | System Integration | Some help by Browsers |
| Maintenance | Reuse of original product | GrabBag, Browsers, Generalizers, Refiners |

Table 1: Reusability Life Cycle Stages vs. Traditional Life Cycle

documented, and verified to satisfy requirement; detailed design specification is the period of time during which the design of system or a system component is documented; typical contents include system or component algorithms, control logic, data structures, data set-use information, input/output formats, and interface description; implementation is the period of time during which a software product is created from design documentation and debugged; testing is the period of time during which the components of a software product are evaluated and integrated to determine whether or not requirements have been satisfied; system integration is the period of time during which a software product is integrated into its operational environment and tested in this environment to ensure that it performs as required;

maintenance is the period of time during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements

A component is a basic part of a system or program; an interface is a shared boundary to interact or communicate with another system component [Sta83]. Glue code is the minimal extra code that may be needed to bring the reused modules together.

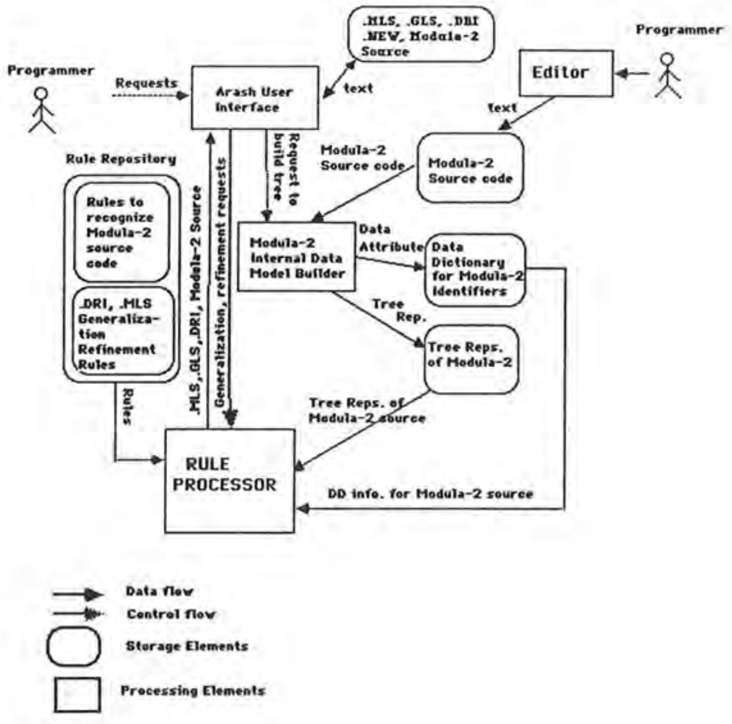
## 2 Arash System Architecture

The overall structure of Arash is shown in Figure 2. A programmer interacts with Arash through requests which are processed by the Arash User Interface Routines (AUIR). These routines provide a user interface (windows, icons, menus, and a mouse), exercise control over the activation of Generalizers and Refiners, and communicate with the Rule Processor. Figure 3 shows what the user interface looks

Figure 2: Arash System Structure

like when running Arash.

A Modula-2 source program is read from a text file by the Modula-2 Interface Data Model Builder ( shown in Figure 2) following a user's request to build an internal tree representation of the Modula-2 text. Symbol table information is stored in the Data Dictionary, and the transformed source program file is stored in the Tree Representation of Modula-2 data structure. This internal data structure is processed by the Rule Processor.



Programmer
Requests
Arash User Interface
.MLS, .GLS, .DRI .NEW, Modula-2 Source
text
Editor
Programmer
text
Request to build tree
Modula-2 Source code
Modula-2 Source code
Rule Repository
Rules to recognize Modula-2 source code
.DRI, .MLS Generalization Refinement Rules
.MLS,.GLS,.DRI, Modula-2 Source
Generalization, refinement requests
Modula-2 Internal Data Model Builder
Data Attribute
Data Dictionary for Modula-2 Identifiers
Tree Rep.
Tree Reps. of Modula-2
Tree Reps. of Modula-2 source
Rules
RULE PROCESSOR
DD info. for Modula-2 source
Data flow
Control flow
Storage Elements
Processing Elements

RULE

Fig 2

Figure 3: Arash User Interface Options

When a user requests that the Tree Representation of a Modula-2 program be generalized, a collection of AUIR's are activated which traverse the tree and produce .MLS, .DRI, and .GLS files. The Rule Processor takes a rule from the Rule Repository, processes it, calls the appropriate AUIR, and outputs the result to the .MLS, .DRI, and .GLS attribute files.

Similarly, when the user requests that the .GLS file of some program be refined, the Refiner uses rules from the Rule Repository to carry out a refinement. To understand how Arash works, one must understand three central structures:

- The Internal Tree and Data Dictionary containing Modula-2 source program data,
- The Rule Repository , Rule Processor, and Syntactic and Semantic structure of Rules,
- The interaction among Rules, Rule Processor, and the Internal Tree/Data Dictionary.

The Rule Processor returns the result of generalization and refinment (.GLS, .MLS, .DRI, and .NEW) to AUIR which in turn displays each file in a text editing window so it may be inspected and saved by the user. These files are described below:

## .DRI Attribute File

The *.DRI* attribute file contains the rules used by the rule processor to transform a source program into an abstraction. This file is created by Generalizers and used

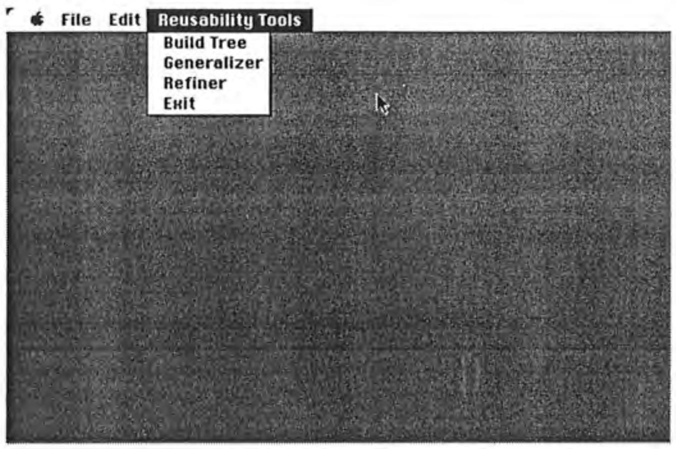

File
Edit
Reusability Tools
Build Tree
Generalizer
Refiner
Exit

Fig3

for refining a module to a concrete form. The rules in a .DRI file are copied into the rule repository by AUIR prior to refining an abstracted module to a concrete one.

## .MLS Attribute File

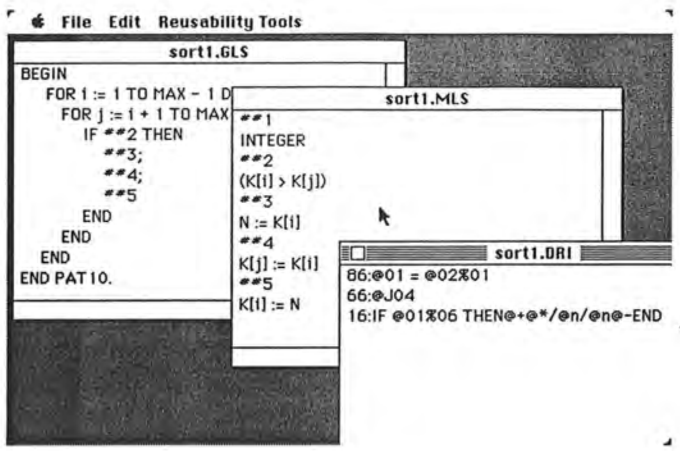
The .MLS attribute file is a text file generated by the rule processor during the generalization of a source program. It contains a list of meta identifiers and the actual source code that each meta identifier replaced. For example, see Figure 4. The .MLS file is used by the Refiner to replace meta identifiers with actual source

Figure 4: Attribute Files Generated by Generalizers

code. The contents of a .MLS file may be modified either by a rule, or by manual editing of the file prior to Refinement. Such modifications allow for restructuring of a reusable module to tailor it to a new purpose.

## .GLS Attribute File

For each source program module which is abstracted by the Generalizers an attribute file .GLS is created which contains a textual representation of the abstracted module. It serves as visual feedback to the user to verify the operation of the Generalizers and has no other significance. The Refiners operate directly on the tree representation of the original input.



sort1.GLS

```
BEGIN
   FOR i := 1 TO MAX - 1 D
     FOR j := i + 1 TO MAX
        IF **2 THEN
           **3;
           **4;
           **5
        END
     END
   END
END PAT10.
```

sort1.MLS

```
**1
INTEGER
**2
(K[i] > K[j])
**3
N := K[i]
**4
K[j] := K[i]
**5
K[i] := N
```

sort1.DRI

```
86:@01 = @02%01
66:@J04
16:IF @01%06 THEN@+@*/@n/@n@-END
```

Fig 4

## Internal Tree Structure

All inputs to Arash are first converted to a tree structure by the Modula-2 Internal Data Model Builder as shown in Figure 2.

Figure 5 shows a Modula-2 source code program as it is converted into a tree structure by the Internal Data Model Builder.

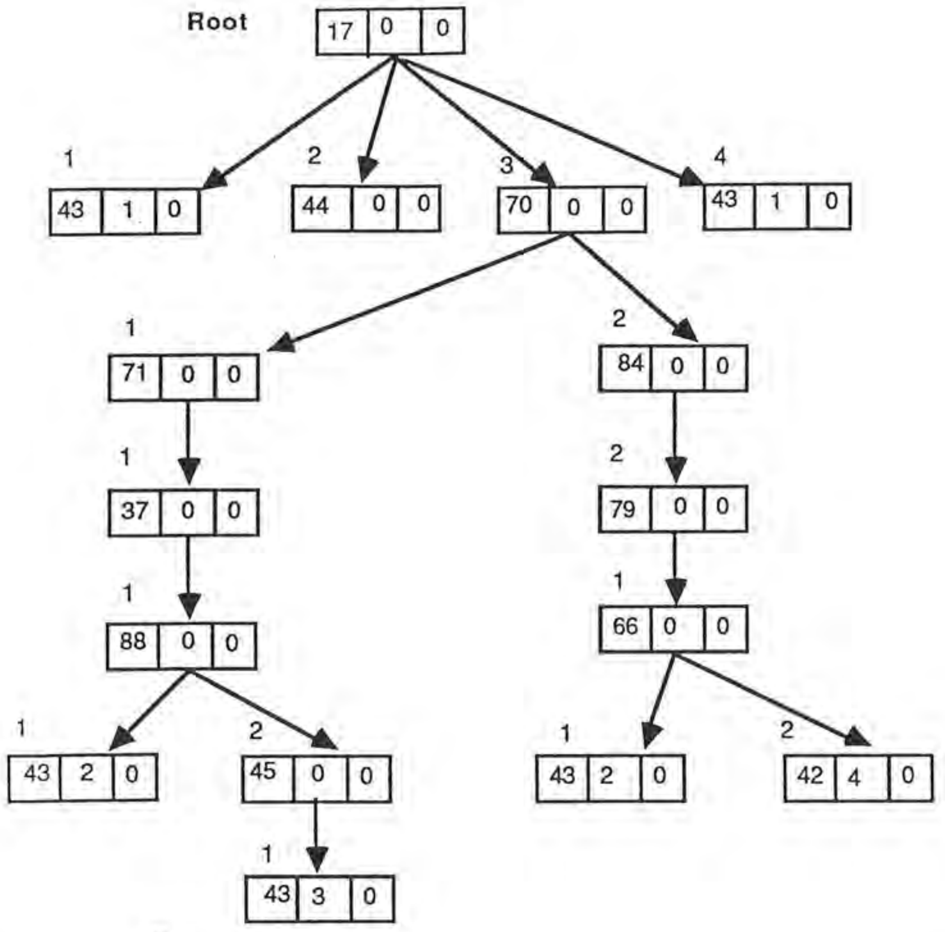
Figure 5: Internal representation of an object in Arash

Each node of the tree holds four categories of information:

**Label Info.:** Each node is assigned a label which indicates its type. Type Information is used by the rule processor to decide what rule to apply to the values stored at each node. In Figure 5 node types 17, 43, 70, ... refer to different logical parts of a typical Modula-2 source program. Note that the tree representation of the sample has some extra node types (45, 43 second subtree of node 88) not usually found in an abstract syntax tree.

**Data Dictionary Reference Pointer:** The data dictionary captures and disseminates information related to attributes of the nodes of the tree. Figure 5 shows the data dictionary of the sample Modula-2 source program. The scope of the main module is assumed to start from 1, so the scope value for **W** will be 1 in the data dictionary.

**Link Information:** Link information maintains the internal representation of a tree. Children of each node are numbered sequentially from the left to right. This *sequence number* is used to access a child of a node. For example, in Figure 5 node label 17 has four children that are numbered from left to right



Data Dictionary

| | |
|---|---|
| 1 | PAT12A (43) |
| 2 | W (43) → 1 |
| 3 | INTEGER (43) |
| 4 | 0 (42) |

(n) = node type
→ m = scope

Legend

| n | | |
|---|---|---|
| label | DDP | ADL |

n: Node sequence number
label: Node Type
DDP: Data Dictionary Reference Pointer
ADL: Application/Data Linkage

DDP = 0 no Data Dictionary Reference
ADL = 0 no Application/Data Linkage

```
IMPLEMENTATION MODULE PAT12A;

  VAR W : INTEGER;
BEGIN

  W := 0;

END PAT12A.
```

17: implementation
37: list of variable declarations
42: number
43: id
44: ;
45: :
66: assignment statement
70: block
71: declaration part
79: list of statements
84: body of module
88: first variable

7

[illegible] Figs 5 [illegible]

as 1 (node 43), 2 (node 44), 3 (node 70), and 4 (node43). Internally, the n-ary tree shown in Figure 5 is maintained as a binary tree based on the natural transformation in [Knu73].

**Application data/linkage:** Each node of the tree contains an application data/linkage field that is used to either hold user-defined values or extend the data structure of each node without altering other portions of the system. In Arash this field has not been used.

## Rule Repository

The *Rule Repository* is the storage element where rules are stored and accessed by the rule processor. This storage element is divided into two logical parts. The first part contains rules to recognize Modula-2 source programs and the second part stores Generalization and Refinement Rules.

### Rules To Recognize Modula-2 Source Programs

These rules reconstruct Modula-2 source programs from the tree representation of Modula-2 source programs. These are copied into the rule repository by the rule processor from *Arash.DRI* file prior to processing the tree representation. See Appendix B for a list of these rules.

### Generalization and Refinement Rules

Generalization of Modula-2 source programs and the refinement of the abstracted modules to concrete ones are done by a set of rules that are either generated by

Generalizers, or supplied by the user through creation or modification of .DRI or .MLS files. Appendix A lists functions which produce generalization and refinement rules, automatically. Manual modification of .DRI or .MLS files results in rules for specific transformations, also.

## The Rule Processor

The *Rule Processor* is a transformational unit which converts Modula-2 source code stored in the tree representation into either Generalized or Refined output. Figure 6 shows the execution environment of the rule processor. Rules direct the rule processor to perform transformational operations on a Modula-2 source code tree. The transformational operations are directed by rules to either reconstruct a Modula-2 program from its tree representation, or to generalize/refine. The rules to reconstruct Modula-2 programs are always the same and reside in the Rule Repository. The rules for generalization are created by the Generalizers and copied to the Rule Repository, on the fly. The rules for refinement are copied from a .DRI file into the Rule Repository prior to refinement.

Figure 6: Arash's Rule Processor Execution Environment

The refinement rules copied from a .DRI file into the Rule Repository reference special–purpose functions which must be installed by loading them into Arash. A function table is provided for this purpose. *Function Table* is a vector that stores the address of Arash Generalizers and Refiners. These functions are listed in Appendix A.

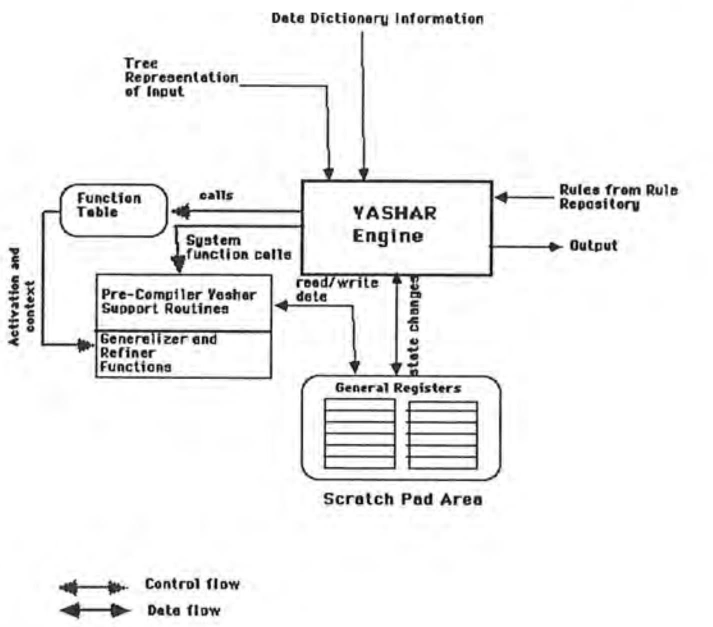

Data Dictionary Information
Tree Representation of Input
Function Table
calls
YASHAR Engine
Rules from Rule Repository
Output
System function calls
Activation and context
Pre-Compiler Yashar Support Routines
Generalizer and Refiner Functions
read/write data
state changes
General Registers
Scratch Pad Area
Control flow
Data flow

Fig 6

## Scratch Pad Area

The Scratch Pad Area is a set of *Registers* used by functions that carry out generalization and refinement to communicate among themselves and the rule processor. Access from within a rule is symbolically referenced as $Rn$ where $n$ is a two digit register number. There are 40 registers *(R01 - R40)* defined for such purposes.

## Rule Processing

The rule processor traverses the tree and processes each node in the tree according to navigational and operational directives specified in each rule. When a labelled node is visited, the repository is searched for a rule with a corresponding label. Then the rule is applied to all tree nodes with the specific label. The next node to be visited is determined by the rule. The minimum sequencing instruction for each rule is @* which causes the tree to be traversed in *depth first* order. To traverse the tree according to the rules stored in the rule repository, the rule processor maintains a *context* for each node. A context consists of:

**Node Priority:** used in generation of parenthesized expressions. There exists a classical problem of regenerating expressions from expression trees when the priority of their operators is altered by using parenthesis [Bro72,CH73,Bro77]. We have modified the solution in [Fri83] in which expression sub-trees are assigned priorities and associativity values. These values provide the capability to decide where to emit parentheses in regeneration of expressions. Arash's solution is a generalization that assigns priorities to a node type rather than the expression node so that the rule processor will emit the proper parentheses. In this new method there is no need to specify the associativity relation of a node.

**Delimiter String Pointer:** is pointer to the location of a common string (e.g.: ;) to be emitted after each child of a node. The rule processor uses the content of the location pointed at by Delimiter String Pointer to fetch the string and output it as a separator of the children of a node. For example, the statements in the body of a loop should be separated by a *;* and a *newline*. Therefore the Delimiter String Pointer will point to *;@n* which the rule processor emits after processing each of the children of that node. The **@D** instruction can be used[BL86] to set the Delimiter String Pointer.

**Pointer to Rule Definition:** refers to the rule definition of a node. The rule for each node to be processed is prefetched prior to its execution and its address is passed to the rule processor.

## Rule Processor Instruction Set

The instruction set of the Rule Processor is divided into the following:

- Tree Navigation
- Formating
- Escape and Breaking
- Register Manipulation
- Miscellaneous

## Rule Syntax

Each rule is a mixture of text, active and passive instructions. To distinguish between instructions and the text that is passed along, instructions are prefixed by an @ symbol.

The components of a rule are:

- a label, always
- one or more active instructions, always
- text, optionally
- one or more formatting instructions, optionally

The label designates the type of node to be operated on by the rule processor. The rule is applied to *all* nodes of the type specified by the label. Active instructions are responsible for, 1) sequencing the processing order of tree nodes, and 2) providing a mechanism for communicating data and control values among the rules and support functions. Formatting instructions and text do not have any effect on tree nodes, and serve only to format the output. For example the following rule:

$$\overbrace{\text{02}}^{\textit{label}} : \underbrace{\mathit{BEGIN}}_{\textit{text}} \quad \overbrace{@n@+}^{\textit{Formatting Inst.}} \quad \underbrace{@M\$R03 = (@01)\$@01}_{\textit{active Inst.}} \quad \overbrace{@-}^{\textit{formatting Inst.}} \quad \underbrace{\mathit{END}}_{\textit{text}}$$

directs the rule processor to transform every node of type 02 as follows:

- Emit a **BEGIN** (BEGIN)
- Emit a newline symbol (@n)

- Increment the indentation level by one increment unit (@+). An *increment unit* is assumed to be four character positions, the default value can only be altered prior to activation of the rule processor.
- Save the address of the first child of the current node in register R03 (@M$R03=(@01)$)
- Process the first child of the current node (@01). To make the rule processor operate on a specific child of a node a child's sequence number is used thus **@01** designates the first child of every node of type 02.
- Decrement the indentation level by one increment unit (@-)
- Emit an **END**

## Tree Navigational Instructions

The order in which children of a tree node are processed is specified by a sequence number prefixed with an @. For example the following causes the rule processor to process first, second, and forth children of all nodes labeled 20 respectively. Here, (@I03) means to ignore child 03 of node type 20.

$$20 : @01@02@I03@04$$

## Formatting Instructions

Formatting instructions are for prettyprinting the textual representation obtained from the Modula-2 tree representation. These instructions do not effect the state of the nodes of a tree. For example @n, @+, and @- cause the rule processor to emit the control sequence to generate a new line, increment indentation level, and decrement the indentation level respectively.

$$02: BEGIN \quad \overbrace{@n@+}^{Formating\ Inst.} \quad @01 \quad \overbrace{@-}^{Formating\ Inst.} \quad END$$

The rule above causes the rule processor to do the following:

- Emit **BEGIN** (BEGIN)
- Emit a newline symbol (@n)
- Increment the indentation level by one increment unit (@+)
- Process the child number one (@01)
- Decrement the indentation level by one increment unit (@-)
- Emit an **END** (END)

## Escape and Break point Instruction

The % symbol designates an *escape* instruction. If a navigation instruction has an % appended to it, the rule processor will execute the function referenced by the next two digits instead of processing the node referenced by the navigation instructions. The two digit number following % is an index into the Function Table which selects the support function to be executed. For example the following directs the rule processor to skip the second child of every node whose label equals 34 and to pass control and context of the rule processor to the 5th function referenced in the Function Table.

34 : @01@02%05

The **@Jn** instruction directs the rule processor to execute the *nth* function in the function table. In contrast to **(%n)**, the context information is not passed to the called function. A list of these functions is given in Appendix A. The rule processor supports the insertion (definition) , activation, and removal of break points to temporarily interrupt the processing of a rule. One can use the break point facility to step through a class of tree nodes, for example. Definition, activation and removal of break points are described in [BL86].

## Arithmetic and Relational Instructions

Arithmetic operations use Scratch Pad registers and constant values. The binary arithmetic operators +, -, *, /, and relational operators `==`, `>=`, `<`, `<=`, `!=` are supported.

$$@M\$R03 = (R02 + R07)\$$$

This rule assigns the sum of the values stored in register two and seven to register three. The precedence and order of evaluation is the same as for the C language [KR78].

## Miscellaneous Instructions

The instructions to manipulate the rule repositories, access the data dictionary information, etc. belong to this category. For example:

$$30 : @m16\$IF@01\%12THEN@ + @D/; @n/@ * @n@ - END\$\ldots$$

will cause the rule processor to first modify the rule definition 16 to what is enclosed between two $ delimiters, and then continue with the remainder of rule 30. To restore the original definition of rule 16, some rule must contain the following ...@r16.

One important miscellaneous instruction is the *if–then–else* instruction used to select one of two alternative actions. The format of this instruction is

$$< label >: @? < test >? < true\ part >? < false\ part >$$

Anytime the rule processor encounters the rule above, it performs the $< test >$, first. If the result is ***true*** then the ***true part*** is used in modifying another rule, otherwise the *false part* is used for modifying a rule. For example:

86:@?( J02 == 1)??50:(StrCmp(@01,@02) = 0 )@m66$StrCpy(@01,@02)@r66$?

- call function number 2 in Function Table (J02)
- if returned result is 1 do the true part, in this case nothing will happen because the true part is empty (??)
- otherwise do the else part which in this case will modify rule 50 to:

(StrCmp(@01,@02) = 0 )@m66$StrCpy(@01,@02)@r66$

# 3 Generalizers and Refiners

Generalizers transform a Modula-2 source code component into a parameterized form called an abstract module. Refiners operate on the abstracted module to produce a concrete instance. Generalizer-Refiner pairs are inverse transformation operators.

A *program fragment* is a piece of source code representing the stereo-typical action sequence in programs [SE83]. Program fragments are open pieces of source code that are meant to be modified or tuned to the particular task at hand [SE83]. For example a *WHILE* loop in a sort routine can be considered a loop fragment.

An abstract module is one in which certain *program fragments* are abstracted into a generic version by substituting a special identifier, called a *meta identifier* in the place of the program fragment. A meta identifier is an identifier prefixed by ## to distinguish the meta identifiers from concrete program identifiers. A concrete module is one in which meta identifiers are replaced by user defined or refiner generated text.

## Generalizer Operation

A Generalizer transforms a set of program fragments $p_i \in P$ into a set of meta identifiers, $q_i \in Q$, where:

**P** Modula-2 source code module

$p_i$ Modula-2 fragment

**Q** Abstracted module

$q_i$ Modula-2 meta identifier

The transformation $G(P) \Rightarrow Q$ carried out by Arash Generalizers re-writes program **P** into meta-program **Q** through a series of generalizing functions:

$$G(P) = g_1 \cdot g_2 \cdot g_3 \cdots g_k(P)$$

The generalization functions $G(P) = g_1 \cdot g_2 \cdot g_3 \cdots g_k(P)$ are generated by the Arash User Interface Routines before they are applied to program fragments to create the abstracted module. The generated rules and the tree representation of source program P are passed to the rule processor where the instructions present in each rule perform the generalization.

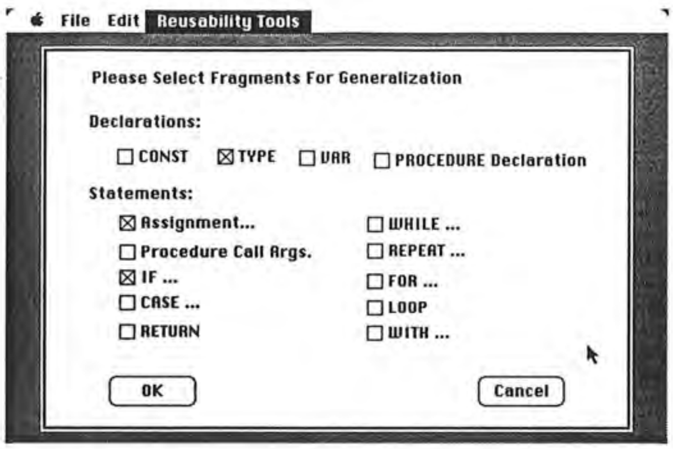
G(P) produces three kinds of output: 1) .GLS attribute file with the abstracted Modula-2 reusable module containing *meta identifiers* in place of fragments, 2) .MLS attribute file containing a list of *meta identifiers* and the actual code that they replaced , and 3) .DRI attribute file containing the rules that were used to generalize P. Figure 4 is an example of these attribute files for the sort routine shown in Figure 1.

A user selects program fragments for generalization from a dialog, for example in

Figure 7: Dialog For Selection of Fragments

Figure 7, a user has selected all *TYPE, ASSIGNMENT*, and *IF* fragments to be generalized. In Figure 8 the user further limits the generalization of *IF* primes to their *conditional* parts. Appendix C lists all menus used for generalization.

Figure 8: Dialog For Generalization of IF



Please Select Fragments For Generalization

Declarations:

[ ] CONST [✓ TYPE] [ ] VAR [ ] PROCEDURE Declaration

Statements:

[✓ Assignment...] [ ] WHILE ...
[ ] Procedure Call Args. [ ] REPEAT ...
[✓ IF ...] [ ] FOR ...
[ ] CASE ... [ ] LOOP
[ ] RETURN [ ] WITH ...

Fig 7

Next, the user's selections are translated to a set of rules which define the functions $G(P) = g_1 \cdot g_2 \cdot g_3 \cdots g_k(P)$. For example the rule in Figure 8 for generalizing the conditional part of *IF* fragments directs the rule processor to do the following anytime it encounters a tree node of type 16 while traversing the tree.

```
16:IF @01%06 THEN@+@D/;@n/@*@n@-END
```

- Emit an **IF** (IF)
- Instead of processing the conditional part (@01) call function number 6 in the Function Table (**%06**) and pass the *context* to it. Function number 6 in the Function Table is *MtIfG()* which is responsible for generating the meta identifier for the conditional part of any IF statement in the Modula-2 source program, and then saving the actual replaced code along with it's meta identifier in the .MLS file. See Appendix A for a list of functions which can be referenced through the Function Table.
- Emit a **THEN** (THEN)
- Increment the indentation level by one increment unit (@+)
- Set the delimiter string to **;@n** (@D/;@n/)
- Process all the children of the node (@*)
- Decrement the indentation level by one increment unit (@-)
- Emit an **END** (END)

## Refiner Operation

A refiner transforms a set of meta identifiers $q_i \in Q$ into a set of program fragments $p_i \in P$ where:

**P** Modula-2 source code module

$p_i$ Modula-2 fragment

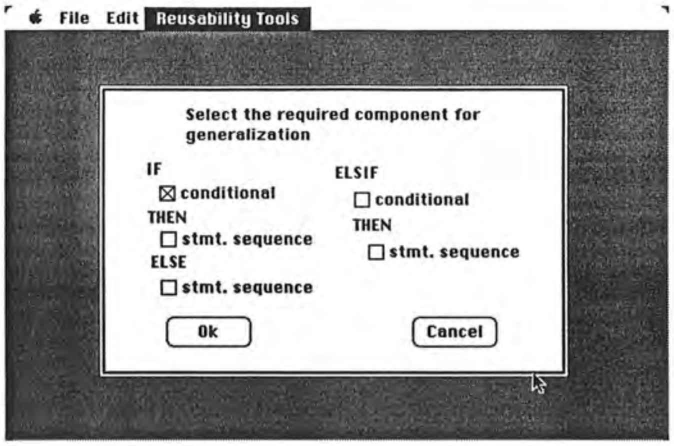
**Q** Abstracted module

$q_i$ Modula-2 meta fragment

The transformation $R(Q) \Rightarrow P$ re-writes an abstracted module **Q** into a concrete program **P** through a series of Refining functions:

$$R(Q) = r_1 \cdot r_2 \cdot r_3 \cdots r_k(Q)$$

The $r_{i's}$ are rules stored in .DRI and .MLS attribute files. Arash copies these rules from the .DRI file into the Rule Repository prior to activation of the rule processor. The rule processor reads the rules stored in the Rule Repository, and the tree stored in the Tree Representation of Modula-2 sources data structure, and writes a refined version of the module into the .NEW attribute file. Furthermore, the .MLS file provides extra flexibility for the refinement process. If at the time of refinement the .MLS file of the abstracted module exists, the meta identifiers in a .MLS file provide the Refiners with: 1) a check point to see if the transformation should be done for the fragment replaced by that meta identifier or not, and 2) check to see if any new rules should be modified in the Rule Repository before the refinement process for the fragment specified by that specific meta identifier. Deletion of any of the



Select the required component for generalization

IF
☒ conditional
THEN
☐ stmt. sequence
ELSE
☐ stmt. sequence

ELSIF
☐ conditional
THEN
☐ stmt. sequence

meta identifiers in the .MLS file has the effect of disabling the refinement process for that specific source fragment. For example, the following will cause the rule 15 to be redefined prior to execution of the *Refiner* which operates on the generalized fragment represented by **##1**.

$$\overbrace{\#\#1}^{\textit{meta identifier}} \quad : \underbrace{15 : @I01@02\%10@J02}_{\textit{Rule to be replaced}}$$

Notice this modification is done on the fly by a Refiner while the rule processor is active and is done in addition to the modifications made prior to activation of the rule processor.

Consider the sort1.mod routine in Figure 1 which has been generalized and then refined into a routine to sort an array of character strings. The comparison and assignment operations are different for integers and strings, so the following rule in the .DRI file converts integer operations into string operations, see Figure 9.

86:@?( J02 == 1)??50:(StrCmp(@01,@02) = 0 )@m66$StrCpy(@01,@02)@r66$?

When the rule processor encounters a node of type 86, (type declaration for array elements), it activates function number 2 (J02) which returns a 1 to specify an integer type and 0 to specify a character string type. If the evaluation of (J02 == 1) is true, meaning that on integer sort is desired, then no rule is modified, otherwise rule 50 is modified to:

(StrCmp(@01,@02) = 0 )@m66$StrCpy(@01,@02)@r66$

The modification of rule 50 causes the generation of the proper comparison construct for character strings and also modifies the assignment rule ( 66) to generate the correct constructs for character strings. See [BL86] for more details on the semantics of the rules. Figure 10 shows the generated sort routine which resulted from using the refinement rule above.

Figure 9: Rules For Re-Structuring Sort Fragment

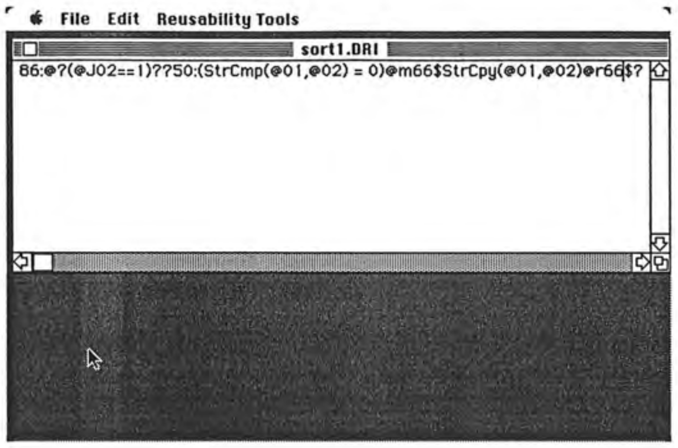
Figure 10: Re-Structured New Sort Fragment

## 4 Conclusion

Arash was built as an experimental tool to study reusability of software systems. Major goals of this effort were: 1) to use existing software components available in existing libraries of source code, 2) avoid creation of new programming languages and notations that are radically different from the majority of current software systems [Weg83,LM83] , namely existing block structured languages, which would discourage application of Arash, and 3) follow the philosophy in which the creation of new software must occur automatically using notation which can be easily generated by computers.

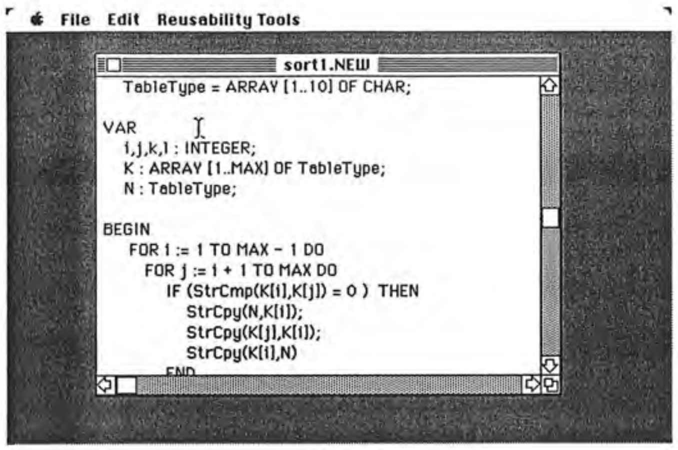
Arash meets all of the goals: 1) it operates on a block structured language, 2) no new programming language is created, and 3) the rule based expressions for restructuring are easily generated and processed by computer.

Access to data dictionary information, flexibility of modifying rules interactively, and two escape mechanisms for semantic processing provide all the necessary tools for deriving a family of concrete programs from a single abstract program.



```
86:@?(@J02==1)??50:(StrCmp(@01,@02) = 0)@m66$StrCpy(@01,@02)@r66$?
```

Fig 9



```
  TableType = ARRAY [1..10] OF CHAR;

VAR
  i,j,k,l : INTEGER;
  K : ARRAY [1..MAX] OF TableType;
  N : TableType;

BEGIN
   FOR i := 1 TO MAX - 1 DO
     FOR j := i + 1 TO MAX DO
       IF (StrCmp(K[i],K[j]) = 0 ) THEN
          StrCpy(N,K[i]);
          StrCpy(K[j],K[i]);
          StrCpy(K[i],N)
       END
```

Fig 10

A limitation that is inherent to the class of languages Arash supports (with the exception of Ada[DOD82]) is the difficulty of mapping the implementation of algorithms that use drastically different data structures. For example, sorting a list of numbers stored in an array versus a linked list. This problem is due to the algorithmic differences between indexing through elements of an array and visiting elements of a linked list. Currently, Arash is not capable of restructuring an array dependent algorithm into an equivalent linked list dependent algorithm.

A related project in reusability through program transformation is described in [Che83]. In that effort an extended version of EL1 programming language is used as the base language. The EL1 language supports programmer–defined data types, generic routines, and programmer control over type conversion [Weg74]. The abstracted programs are built in EL1 and the transformation is done by defining transformational rules containing a syntactic pattern part , optionally augmented by a semantic predicate and a replacement. The main advantage of Arash is it's applicability to more than one language in a family of languages and its support for existing software programs.

# A Generalizer and Refiner Functions

Reusability functions are divided into two groups: 1) functions for Generalization, and 2) functions for Refinement. There is a generalization and refinement support function defined for each language fragment. These functions are installed in the Function Table by the Arash User Interface Routines when Arash is started. The reference index to each function is shown in front of each function name.

The context of the rule processor, a pointer to the rule definition, and a pointer to the tree node that will be generalized or refined are passed to the function when activated by the Rule Processor. These functions are assumed to return a pointer to a character string as the result of their activation. If nothing is to be returned, a null pointer is returned.

A user can alter the semantics of each of the generalization and refinement functions by installing his own.

## A.1 Generalization support Functions

As part of their activities each of these functions produce the meta identifiers for the constructs that they support.

**0 MtConsG():** Generalization function for constant fragments.

**1 MtTypeG():** Generalization function for type fragments.

**2 MtVarG():** Generalization function for variable declaration fragments.

**3 MtPrcG():** Generalization function for procedure declaration fragments.

4 **MtAssG():** Generalization function for assignment statement fragments.

5 **MtPCallG():** Generalization function for procedure call fragments.

6 **MtIfG():** Generalization function for if fragments.

7 **MtCaseG():** Generalization function for case fragments.

8 **MtWhileG():** Generalization function for while fragments.

9 **MtRepeatG():** Generalization function for repeat call fragments.

10 **MtForG():** Generalization function for for fragments.

11 **MtLoopG():** Generalization function for loop fragments.

12 **MtWithG():** Generalization function for with fragments.

13 **MtReturnG():** Generalization function for return fragments.

## A.2 Refinement Support Functions

The refinement support functions operate on the abstracted fragments to create a concrete instance. If the .MLS file exists for the abstracted module under refinement its functionality is extended to perform extra steps as explained in Refiner Operation section.

**0 CuConsG():** Refinement function for constant fragments.

**1 CuTypeG():** Refinement function for type fragments.

**2 CuVarG():** Refinement function for variable declaration fragments.

**3 CuPrcG():** Refinement function for procedure declaration fragments.

**4 CuAssG():** Refinement function for assignment statement fragments.

**5 CuPCallG():** Refinement function for procedure call fragments.

**6 CuIfG():** Refinement function for if fragments.

**7 CuCaseG():** Refinement function for case fragments.

**8 CuWhileG():** Refinement function for while fragments.

**9 CuRepeatG():** Refinement function for repeat call fragments.

**10 CuForG():** Refinement function for for fragments.

**11 CuLoopG():** Refinement function for loop fragments.

**12 CuWithG():** Refinement function for with fragments.

**13 CuReturnG():** Refinement function for return fragments.

## B Rules To Recognize Modula-2 Source

These rules are needed to reproduce the original Modula-2 source program text from the Tree Representation in main memory. In some cases, no rule is needed, in which case the null rule **NA** is used.

```
0:
1:ARRAY @D/,/@01 OF @02
2:***NA***2
3:BY @01
4:CASE @01 OF @n@+@D/ |@n/@02@n@+@03@-@-@nEND
5:CONST @D/;@n/@n@+@*//;@n@-
6:DEFINITION MODULE @01;@n@*//.@n
7:@01 DIV @02
8:DO @n@+@D/;@n/@01
9:@-ELSE@n@+@D/;@n/@01@-@+
10:@-ELSIF @01 THEN@n@+@D/;@n/@02@*/;@n/@-@+
11:END
12:EXIT
13:EXPORT @D/,/@01;
14:FOR @01 := @02 TO @03 @04 @05 @-@nEND
15:@+FROM @01@02@n@-
16:IF @01 THEN@+@*/@n/@n@-END
17:IMPLEMENTATION MODULE @01@02@*//.@n
18:@+IMPORT @D/,/@01;@n@-
19:@01 IN @02
20:LOOP@n@+@D/;@n/@01@-@nEND
```

```
21:@01 MOD @02
22:MODULE @*//
23:NOT @01
24:***NA***24
25:@01 OR @02
26:***NA***26
27:PROCEDURE @01@*//
28:QUALIFIED @D/,/@01;
29:@n@+RECORD @n@+@D/;@n/@*//@n@- END@-
30:REPEAT @n@+@D/;@n/@01@n@-UNTIL @02
31:RETURN @*//
32:{@D/,/@01}
33:@D/;@n/@01
34:***NA***34
35:TYPE@D/;@n/@n@+@01;@n@-
36:***NA***36
37:VAR @D/;@n/@n@+@*//;@n@-
38:WHILE @C+@01@D/;@n/@C- DO @+@n@02@n@-END
39:WITH @01 DO @n@+@D/;@n/@02@-@nEND
40:@01
41:"@d"
42:@d
43:@d
44:;@n
45:***NA***45
46:***NA***46
47:@01.@02
```

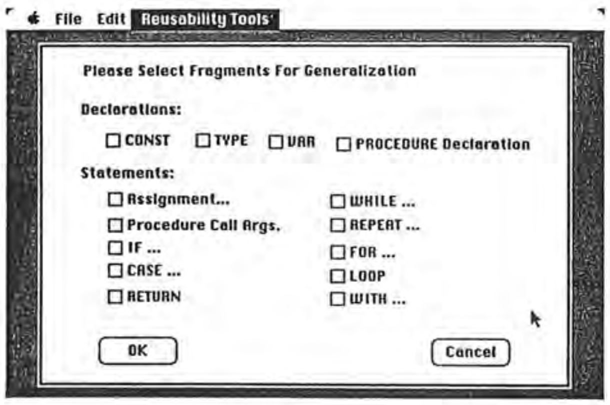
```
48:@01..@02
49:@Cm(@01 < @02)@Cm
50:(@01 > @02)@Cm
51:(@01 = @02)@Cm
52:(@01 >= @02)@Cm
53:(@01 # @02)@Cm
54:(@01 <= @02)@Cm
55:@^1@(@01 + @Cm@02@)
56:@^2@(@01 / @Cm@^3@02@^2@)
57:@^2@(@01 * @Cm@02@)
58:@^1@(@01 - @Cm@^2@02@^1@)
59:@Cm@01 & @02@Cm
60:***NA*60
61:***NA*61
62:***NA*62
63:***NA*63
64:***NA*64
65:***NA*65
66:@01 := @02
67:***NA*67
68:@01^
69:***NA*69
70:@01@n@02
71:@*/@n/
72:(@D/,/@01)
73:@D/./@01
74:ARRAY@D/,/@01 OF @02
```

```
75:@01@*//
76:(@D/; /@01)@02;@n
77:@D/,/@01:@02
78: [@01];@n
79:@01@*D
80:@01[@02]
81:@^O@(+@01@)
82:@^O@(-@01@)
83:@01 : @n@+@D/;@n/@02@-
84:BEGIN@n@+ @D/;@n/@01@*D@-@nEND
85:@01
86:@01 = @02
87:(@D/,/@01)
88:@D/,/@01 : @02
89:SET OF @01
90:POINTER TO @01
91:@D/,/@01 : @n@+@D/;@n/@02@-
92:[@01..@02]
93:***NA*93
94:@D//:@01
95:@D/./@01
96:@01 : @02
97:@01@*//
98:CASE @01 OF @n@+@D/ |@n/@02@n @03 @-END
99:@01 = @02
100:VAR @D/,/@01@02
101:PROCEDURE @D/, /(@01)@02
```

```
102:VAR @01
103:@D//:@01
104:@-ELSE@n@+@D/;@n/@01@-@+
```

## C Generalizer Selection Dialogs

This section contains dialog boxes used to select language fragments to be Refined.



Please Select Fragments For Generalization

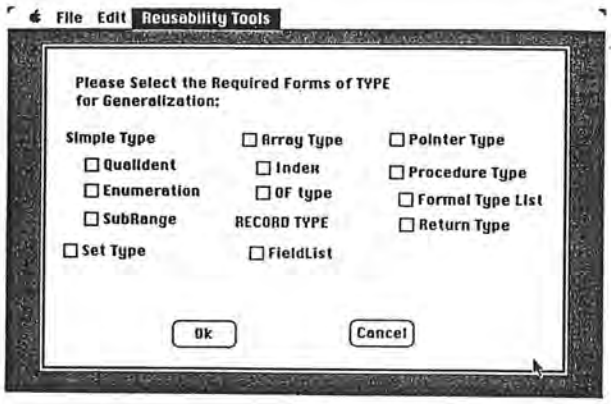
Declarations:

☐ CONST ☐ TYPE ☐ VAR ☐ PROCEDURE Declaration

Statements:

| | |
|---|---|
| ☐ Assignment... | ☐ WHILE ... |
| ☐ Procedure Call Args. | ☐ REPEAT ... |
| ☐ IF ... | ☐ FOR ... |
| ☐ CASE ... | ☐ LOOP |
| ☐ RETURN | ☐ WITH ... |

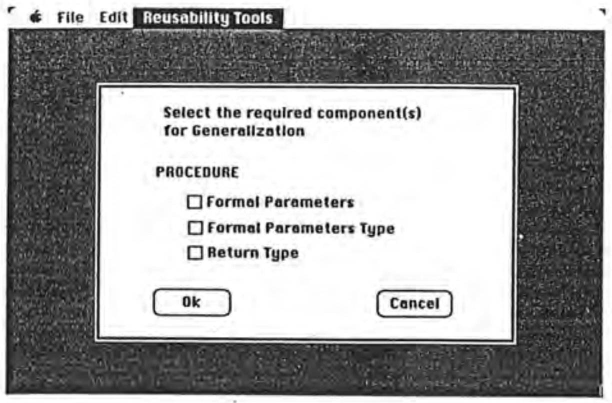
Figure C.1: Selection Dialog for Modula-2 Language Fragments



Please Select the Required Forms of TYPE for Generalization:

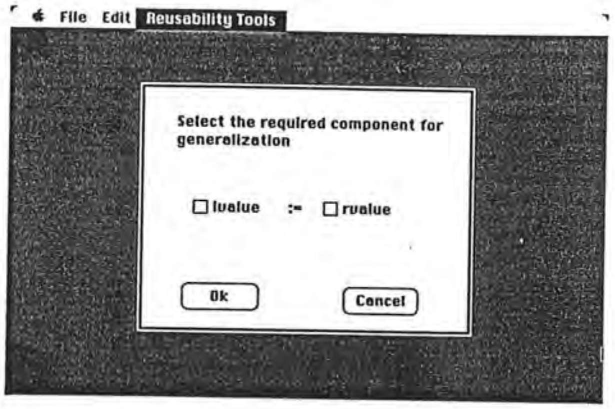
| | | |
|---|---|---|
| Simple Type | ☐ Array Type | ☐ Pointer Type |
| ☐ Qualident | ☐ Index | ☐ Procedure Type |
| ☐ Enumeration | ☐ OF type | ☐ Formal Type List |
| ☐ SubRange | RECORD TYPE | ☐ Return Type |
| ☐ Set Type | ☐ FieldList | |

Figure C.2: Selection Dialog for Type Fragments



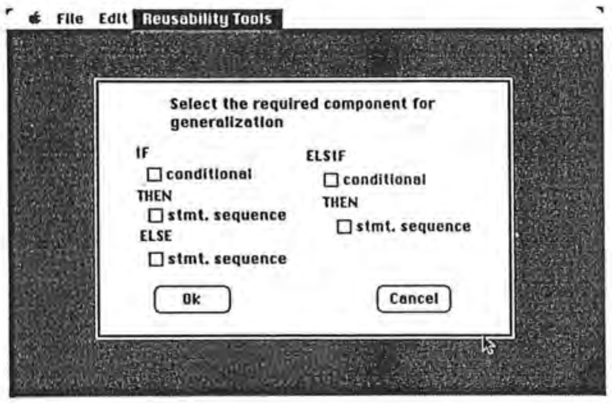
Select the required component(s) for Generalization

Figure C.3: Selection Dialog for Procedure Declaration Fragment



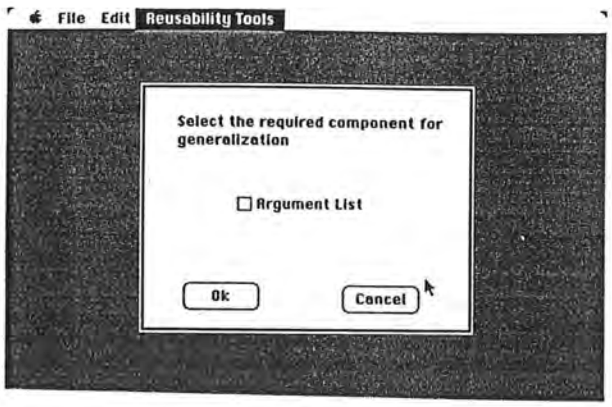
Select the required component for generalization

Figure C.4: Selection Dialog for Assignment Fragment



Select the required component for generalization

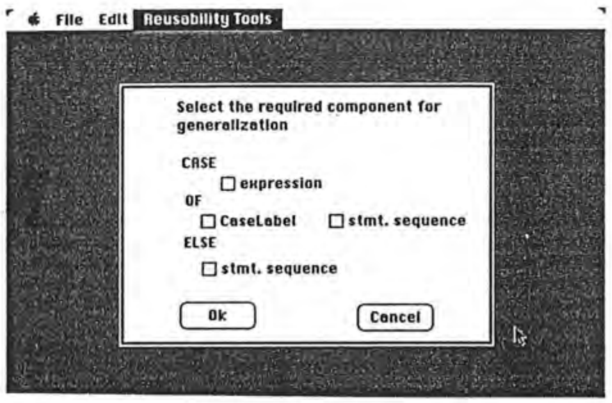
Figure C.5: Selection Dialog for IF Fragment



Select the required component for generalization

☐ Argument List

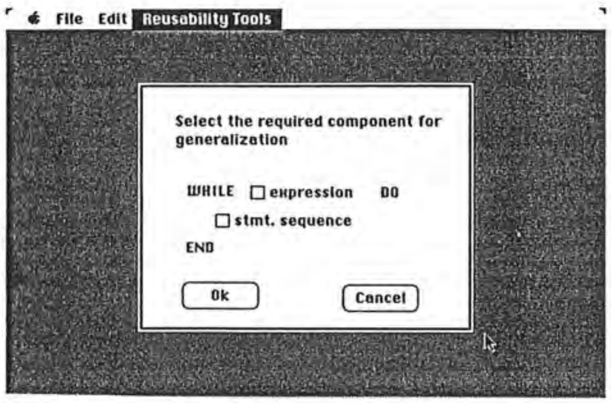
Figure C.6: Selection Dialog for Procedure Call Fragment



Select the required component for generalization

CASE
☐ expression
OF
☐ CaseLabel ☐ stmt. sequence
ELSE
☐ stmt. sequence

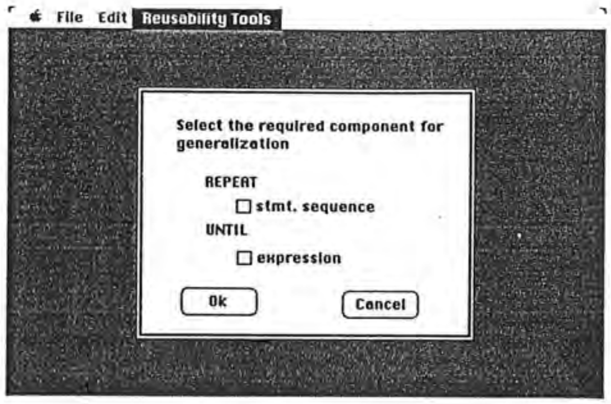
Figure C.7: Selection Dialog for CASE Fragment



Select the required component for generalization

WHILE ☐ expression DO
☐ stmt. sequence
END

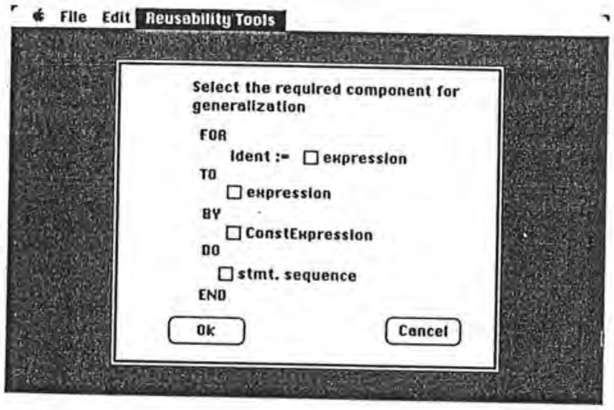
Figure C.8: Selection Dialog for WHILE Fragment



Select the required component for generalization

REPEAT
☐ stmt. sequence
UNTIL
☐ expression

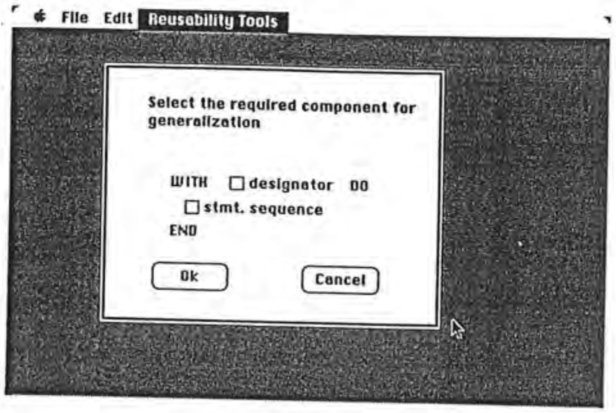
Figure C.9: Selection Dialog for REPEAT Fragment



Select the required component for generalization

FOR
ident := ☐ expression
TO
☐ expression
BY
☐ ConstExpression
DO
☐ stmt. sequence
END

Figure C.10: Selection Dialog for FOR Fragment



Select the required component for generalization

WITH ☐ designator DO
☐ stmt. sequence
END

Figure C.11: Selection Dialog for WITH Fragment

## References

[Bir86] Abbas Birjandi. *A Rule Based Environment for Software Reuse.* PhD thesis, Computer Science Department, Oregon State University, 1986.

[BL86] Abbas Birjandi and T. G. Lewis. *Yashar:A Rule Based Meta-Tool For Program Development.* Technical Report 86-30-6, Department of Computer Science Oregon State University, Corvallis Oregin 97331, 1986.

[Bro72] P. J. Brown. Re-creation of source code from reverse polish form. *Software–Practice and Experience*, 2:275–278, 1972.

[Bro77] P. J. Brown. More on the re-creation of source code from reverse polish form. *Software–Practice and Experience*, 7:545–551, 1977.

[CH73] C. C. Charlton and P. G. Hibbard. A note on recreating source code from the reverse polish form. *Software–Practice and Experience*, 3:151–153, 1973.

[Che83] T.E. Cheatham. Reusability through program transformations. In *Proceedings of Workshop on Reusability in Programming*, pages 122–128, The Media Works, Inc., Newport, RI, September 1983.

[DOD82] DOD. *Reference Manual for the Ada Programming Language.* United States Department of Defense, Washington DC, July 1982.

[Fre83] Peter Freeman. Reusable software engineering:concepts and research directions. In *Proceedings on Workshop on Reusability in Programming*, pages 2–16, Newport, 9 1983.

[Fri83] Peter Fritzson. *Adaptive Prettyprinting of Abstract Syntax Applied to ADA and PASCAL.* Technical Report, Deptartment of Computer Science, Linkoping University, Linkoping, Sweden, September 1983.

[Ker83] Kernighan. The unix system and software reusability. In *Proceedings of the Workshop on Reusability in Programming*, pages 235–239, The Media Works, Inc., Newport, RI, September 1983.

[Knu73] Donald E. Knuth. *The Art of Computer Programming.* Volume 1, Addison Wesley, 2 edition, 1973.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language. Prentice-Hall Software Series*, Prentice-Hall, 1978.

[LM83] S.D. Litvintchouk and A.S. Matsumoto. Design of ada systems providing reusable components. In *Proceedings of Workshop on Reusability in Programming*, pages 198–206, The Media Works, Inc., Newport, RI, September 1983.

[SE83] Elliot Soloway and Kate Ehrlich. What do programmers reuse? theory and experiment. In *Proceedings of Workshop on Reusability in Programming*, pages 184–191, The Media Works,Inc., Newport, RI, September 1983.

[Sta83] American National Standard. *IEEE Standard Glossary of Software Engineering Terminology.* New York, February 1983.

[Weg74] Ben Wegbreit. The treatment of data types in EL1. *Communication of the ACM*, 17(5):251–264, May 1974.

[Weg83] Peter Wegner. Varieties of reusability. In *Proceedings of Workshop on Reusability in Programming*, pages 30–44, The Media Works, Inc., Newport,RI, 9 1983.

[Wir83] Niklaus Wirth. *Programming In Modula-2. Texts and Monographs in Computer Science*, Springer-Verlag, Berlin Heidelberg, 1983.