SOFTWARE CONSTRUCTION USING TYPED FRAGMENTS

Nazim H. Madhavji
Nikos Leoutsarakos
Dimitri Vouliouris

School of Computer Science
McGill University
805 Sherbrooke Street West
Montreal, PQ,
CANADA H3A 2K6

ABSTRACT

Recent research in the field of programming environments has resulted in integrated systems which demonstrate their use in the development of small programs. It is argued here that such systems are not suitable for non-trivial software development, as they support programming-in-the-small only. This paper introduces a new concept of a typed fragment called *fragtype*, which makes the notion of a software building block concrete. With the help of the underlying fragtype driven structured editor, and a fragment library, such building blocks can be used to construct a well-formed large software edifice.

1. Introduction

The concept of integration has recently precipitated widespread research efforts in combining programming tools, such as an editor, compiler, linker and debugger into coherent programming environments. Examples of such systems include the Cornell Program Synthesizer [TeiRe81], ALOE [MedNo81], MENTOR [DHKL84], Magpie [DelMS84], POE [FJMPS84], PECAN [Reiss84] and COPE [ArcCo81]. While such systems have clearly demonstrated their use in the development of syntactically, and in some cases, static-semantically correct *small* programs, their viability is still to be tested in the development of *reasonably large* programs.

It is argued here that currently available program synthesizers are not suitable for non-trivial software development, primarily because they support programming-in-the-small only. For the development of reasonably large programs, a highly integrated scratch pad facility based on a new concept of a fragment type, called *fragtype*, is proposed here.

Fragtypes have a formal basis, similar to data types in Pascal-like programming languages, and therefore they provide protection during the construction of software. A fragment of a certain fragtype can contain objects which are compatible with that fragtype only. Such objects can be of small granularity, such as an expression or they can be of large granularity, such as a subsystem of a program. Thus, a fragment is a formal structure of variable granularity.

In order to manipulate fragments, the scratch pad provides a structured editor which can be used to create a new fragment. The editor also has the capability to develop, refine and assemble existing fragments into a new one, possibly of a different fragtype, in an integrated and well-defined manner. Thus, the editor is a machine for fabricating software from fragments of various fragtypes.

One striking difference between this editor and other structured editors is that the former is driven by fragtypes. Hence, it automatically adjusts itself according to the fragtype of the fragment being operated upon. This is a dynamic feature of the editor, as a fragtype can change at any time depending on the user action. It is this feature of the editor, combined with the concept of fragtypes, which makes the scratch pad flexible enough to suit wide varieties of software development methodologies and yet provide protection during software construction.

Seven major software engineering notions considered in the design of the scratch pad are:

- Software building blocks.
- Rigorous construction.
- Top-down and bottom-up methodologies.
- Repository for building blocks.
- Integration of activities.
- Testing of building blocks.
- Development tool.

The above mentioned points are a subject of current research in the context of the MUPE-2 project at McGill University. This paper focuses on the scratch pad facility which is an important component of the project. Before considering the scratch pad in more detail, the next section puts it into perspective.

2. The MUPE-2 Environment: An Overview

The McGill University Programming Environment (MUPE-2), is an integrated environment for the design, development and use of Modula-2 [Wirth82] programs. The level of MUPE-2 (see Figure 1) can be viewed as above that of program synthesizers, but beneath that of full software engineering environments, such as CADES [Snowd81], PWB/UNIX [Ivie77], SDS [Alfor81] and others.

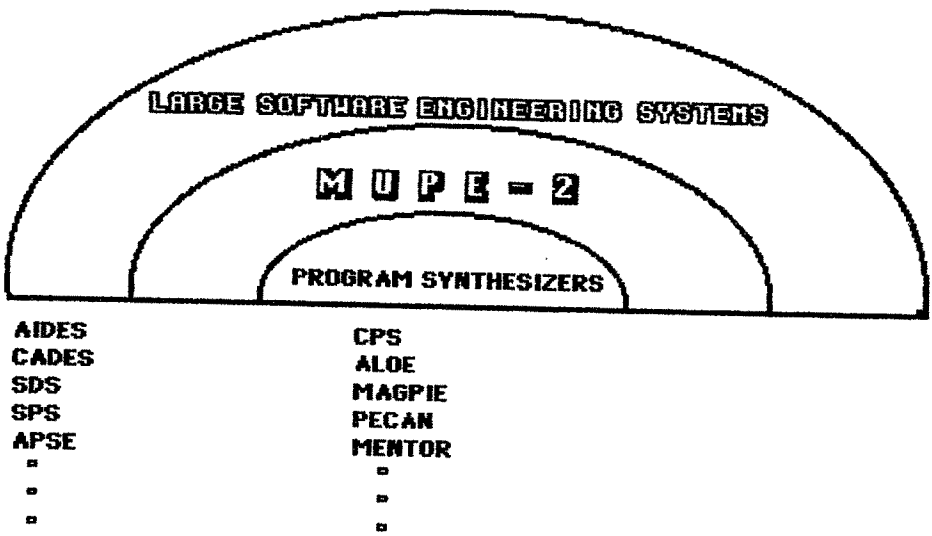

Figure 1 - The level of MUPE-2

MUPE-2 has a characteristic coloured user interface, which is divided into what are termed the module screen, the procedure screen and the scratch pad, as shown in Figure 2. The module screen is used for programming-in-the-large on a chosen implementation module. Here, with the use of its context-sensitive structured editor, a number of operations can be performed on the internal nodes of the module tree. Besides, the module screen can communicate with the scratch pad by transferring subsystem fragments to/from the personal fragment library called FRAGLIB.

The procedure screen is used for programming-in-the-small on a chosen procedure/module (e.g. T) from the current module on the module screen, thus maintaining the complete

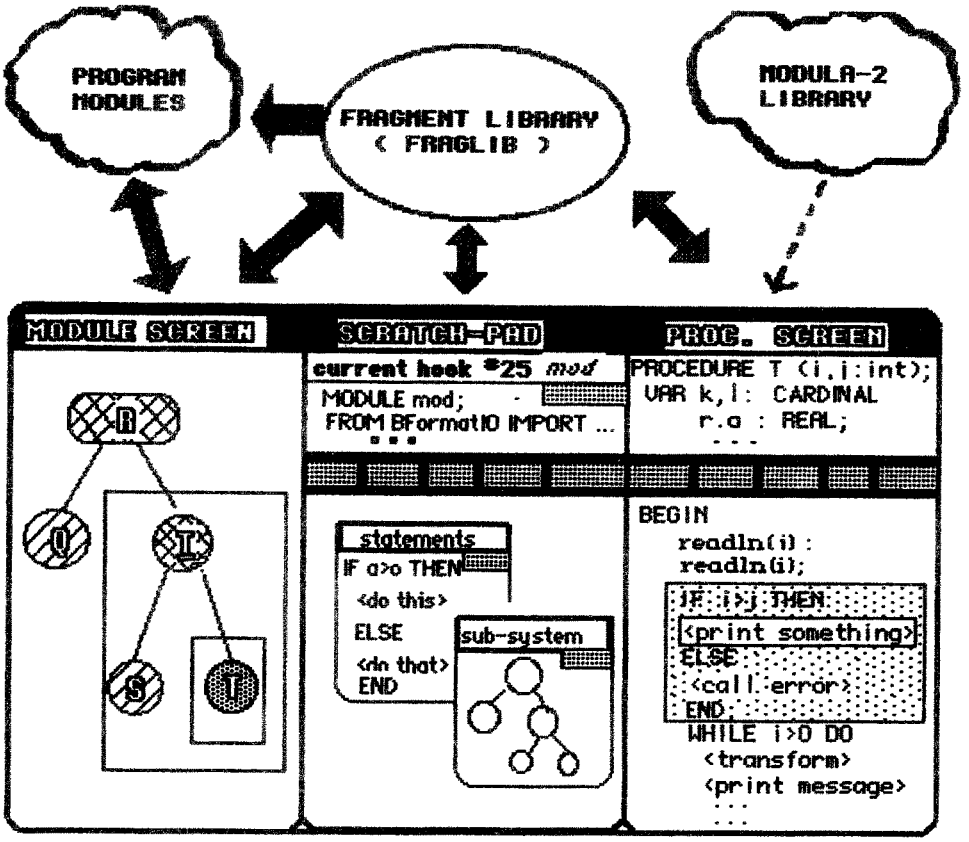

Figure 2 - MUPE-2 screen layout.

environment of the procedure. Operations of the editor permit manipulation of language and meta-language templates and English phrases. Besides, similar to the module screen, the procedure screen can communicate with the scratch pad by transferring procedure fragments to/from FRAGLIB.

The scratch pad is a context-free multi-purpose workbench of the system, where subsystem and procedure fragments may be developed, assembled and tested for inclusion in the main program or in FRAGLIB. Together, the three screens serve the widely known activities of software engineering: programming in-the-large and in-the-small, design, experimentation and testing in a

highly integrated manner.

The key features of MUPE-2 are summarised by the following :

- Its partitioned user interface.
- A scratch pad facility for operating on typed fragments.
- Universal operations based on the structured cursor.
- Coloured graphics for visually (instead of textually) conveying semantic information to the user.
- A number of contextual views, to support display, editing, assembling and execution of subsystem and program fragments.
- Call-tree and user selected walk-through mechanisms.
- Integrated documentation capability based on programming decisions, their refinements and textual or graphical comments.
- Internal representation which is minimal and is compatible with user operations.

3. Software Construction in the Scratch Pad

The scratch pad provides a context-free environment to the user, so that program fragments can be developed independent of the main program. This implies that semantic checking in the scratch pad is performed up to the fragment boundary. In contrast, full semantic checking can be carried out in the procedure and the module screens, since the entire language (Modula-2) environment is available there.

A new fragment can be built in the scratch pad, from scratch, by jotting down ideas as English phrases or by constructing an expression, statements, declarations, a procedure, a module, a system-layer (described later) or a subsystem. This construction is facilitated by the underlying structured editor. For identification purposes, a fragment may be given a name with its description. By default, the system issues an unique fragment number.

If desired, an existing fragment can be selected from a set of working fragments, or it can be unhooked from FRAGLIB, the fragment library. The library is a collection of fragments designed in the scratch pad, hooked fragments of a procedure from the procedure screen and hooked fragments of a subsystem from the module screen.

The underlying editor has the capability to manipulate fragments of different fragtypes, so that they can be developed, refined and assembled into new fragments in an integrated but orderly manner. A new fragment can be hooked into FRAGLIB for later use on any of the three

screens or it can be retained in the scratch pad as one of the working fragments.

3.1. Software Building Blocks

Construction of a reasonably large program generally involves programming in-the-large and in-the-small. During this activity, many utilise both top-down and bottom-up methods of development. However, a system can take a long period to complete, and therefore, rapid prototyping is often desirable to quickly determine the nature of the eventual system. In addition, during the design of such a system, one may experience mundane tasks of re-inventing program structures that are already in use in other projects, and often, one may need to search for efficient and well-written algorithms.

Well-defined software building blocks are a step towards solving the above mentioned problems in software engineering, as they provide formal structures for assembling and re-using software. In MUPE-2, a fragment is a building block, and it is well-defined because it is a fragtyped structure which can be identified through its attributes. A fragtype indicates how the associated fragment can be combined with other structures.

The following list describes the basic form of fragtypes.

Expression: This fragtype contains one expression only.

Declarations: This fragtype contains a sequence of declarations only.

Statements: This fragtype contains a sequence of statements only.

Procedure: This fragtype contains one procedure only.

Module: This fragtype contains one module only.

System-layer: This fragtype contains a combination of procedures, modules and subsystems which have the same parent. For example, in Figure 3, (B, C, D) is a system-layer of node A; whereas, (P, A) is a system-layer for node X.

Subsystem: This fragtype contains a combination of procedures and modules which have a hierarchical relationship. This relationship is structural, shown by the tree arcs, and is according to the target language rules. In addition, the *uses-relationship* is based on procedure calls within a given node, and is dealt with by the incremental semantic analyser. Figure 3 shows that A(B, C, D) and X(P, A) are subsystems, where leaf nodes are treated as procedures or modules as the case may be.

Abstract: This fragtype contains a sequence of English-like phrases only. Each phrase is an abstract representation, at a user chosen conceptual level, of a programming solution. For example, a list of phrases may represent a layer of system modules, a set of declarations, a set of statements, etc. This choice of target objects is a user's decision. MUPE-2 does not *understand* a phrase, as it is not knowledge based. Hence, onus is upon the user to make certain that the phrase is written with intent.

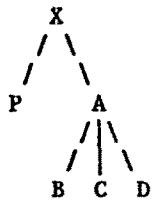

Figure 3 - Subsystem and System-layer relationships

The breakdown of fragtypes above is generalised, in order to avoid specific details of Modula-2. In this language, for example, there can be several kinds of subsystems, such as Implementation-Module-Subsystem, Unit-Subsystem, Procedure-Subsystem and Program-Module-Subsystem. Also, there is richness in fragtypes for data declarations and module interface. In contrast, fragtypes for Pascal are much simpler. This simplicity is reflected in the homogeneity of subsystem and system structures described above. In essence, the concept of fragtypes is powerful enough to be applicable to a class of programming languages.

A parallel can be drawn between fragtypes and Pascal-like data types. Whereas fragments of various fragtypes can be used to construct larger structures such as procedures, modules and subsystems, data items of various types can be used to construct smaller structures such as lists, trees and arrays. In contrast, however, a fragtype is subject to transitions from one fragtype to another.

Figure 4 illustrates the flexibility together with the protection provided by fragtypes and their operations during system construction. For example, it shows that a fragment of fragtype Abstract can be refined into a fragment of another fragtype. This is useful for both programming in-the-small and in-the-large. It also shows that statement and declaration fragments can be turned into procedure and module fragments, say, in bottom-up design. Similarly, procedures and modules can form a system-layer which can then be turned into a proper subsystem. Notice that

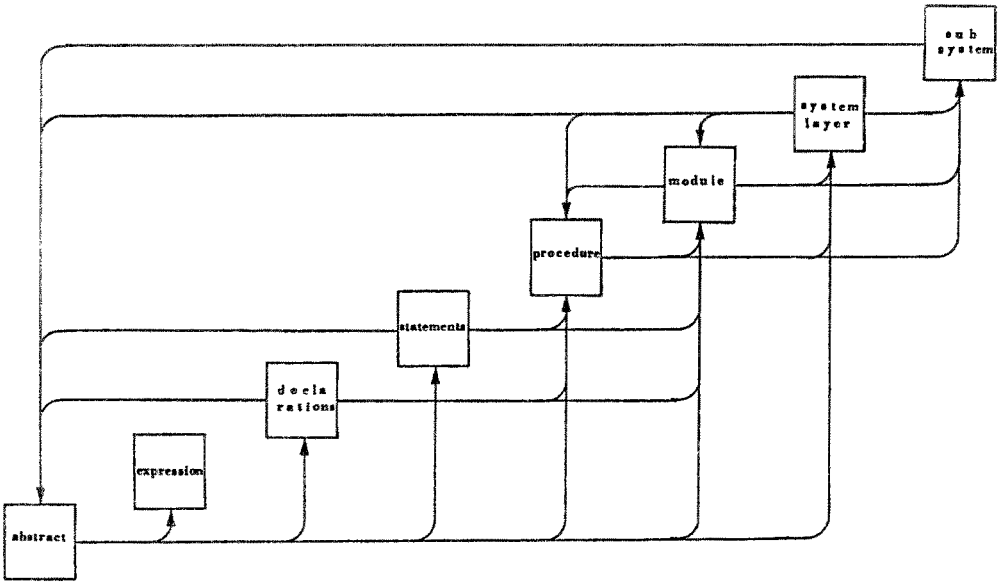

Figure 4 - Fragtype transition diagram

it is also possible to arrive at smaller structures from larger ones, and to transform procedures and modules.

These transitions of fragtypes are achieved by using various commands, such as Copy/Insert, Delete, Transform, Replace/Refine, and their variants. However, before illustrating specific examples of usage, the next section introduces semantic rules which are applied during the fabrication of software.

3.2. Rigorous Construction

Because fragtypes are formal, similar to data types in Pascal-like languages, it is possible to formulate semantic rules to ensure correct fragtype transitions, and fragtype compatibility rules to ensure well-formed fragments.

First, some meta-symbols are introduced so that they can be used in the fabrication rules that follow:

{}n	means >= n times
<...>	means inserted-around
/|\	means which-is-root-of
[]	means optional
|	means or
::=	means fragment-is-composed-of

Subsystem	::=	(Procedure|Module) /|\ (System-layer|Procedure|Module|Abstract)
System-layer	::=	{Procedure|Module|Subsystem|Abstract}2
Module	::=	Module-template <...> [Declarations|Abstract][Statements|Abstract]
Procedure	::=	Procedure-template <...> [Declarations|Abstract] [Statements|Abstract]
Abstract	::=	{English phrase}1
Statements	::=	{statement}1
Declaration	::=	{declaration}1
Expression	::=	expression

These rules ensure that structures are well-formed according to the target language. For example, inserting a fragment of fragtype Declarations in the midst of a fragment of fragtype Statements is not possible. This principle is similar to the data type compatibility rules in strongly typed languages. The benefit here is that a system constructed from basic building blocks is completely well-formed.

It is worth mentioning here that the fabrication rules do not restrict shared use of a component by other components. This is a semantic issue which is resolved by the semantic analyser. In MUPE-2, the user is informed about legal calls to procedures from a given component in a subsystem, with the help of colour coding.

Besides fragment-level semantics, there can be semantic checking within a fragment. For example, in the following fragment of declarations, 'elementtype' is not defined.

Declarations

TYPE
range = 1 .. 10;
a = ARRAY [range] OF elementtype;

This could have been deliberate, as it may already have been defined in the procedure in which this fragment is to be inserted. Therefore, 'elementtype' is highlighted with a colour which means *semantic caution* rather than semantic error. Such checking for a fragment is possible by retaining a local symbol table.

An important point to note is that semantic checking terminates at the boundary of a fragment. This is because the fragment is context-free. All semantic failures in a fragment, which would normally be flagged as semantic errors on the procedure and the module screens, are flagged as semantic cautions on the scratch pad.

Notice that in the case of a newly created fragment of fragtype Statements, all variables are semantic cautions. In the case of a fragment of fragtype Subsystem, checking can be more extensive because simple operations such as insert and delete can have major effects on the rest of the subsystem, in terms of non-local accesses and procedure calls.

At the point of insertion of a fragment in an environment (i.e. another fragment, current procedure or module), incremental semantic checking takes place. If the environment is on the scratch pad then semantic cautions, if any, are highlighted. Otherwise, semantic errors are highlighted.

3.3. Top-down and Bottom-up Methodologies

While the fragtype compatibility rules described in the previous section are rigorous, they do not support any particular development methodology. In particular, providing flexibility of top-down and bottom-up methodologies at any stage of software development is an important asset of a development tool.

The scratch pad provides this flexibility by automatically changing the fragtype of a particular fragment, depending on a user action. Figure 5 shows the operations which can trigger off a fragtype change, and Figure 6 is an example sequence of top-down and bottom-up actions. From this, it is clear that the scratch pad facilitates programming in-the-large and in-the-small, and top-down and bottom-up methods, in an integrated and orderly manner.

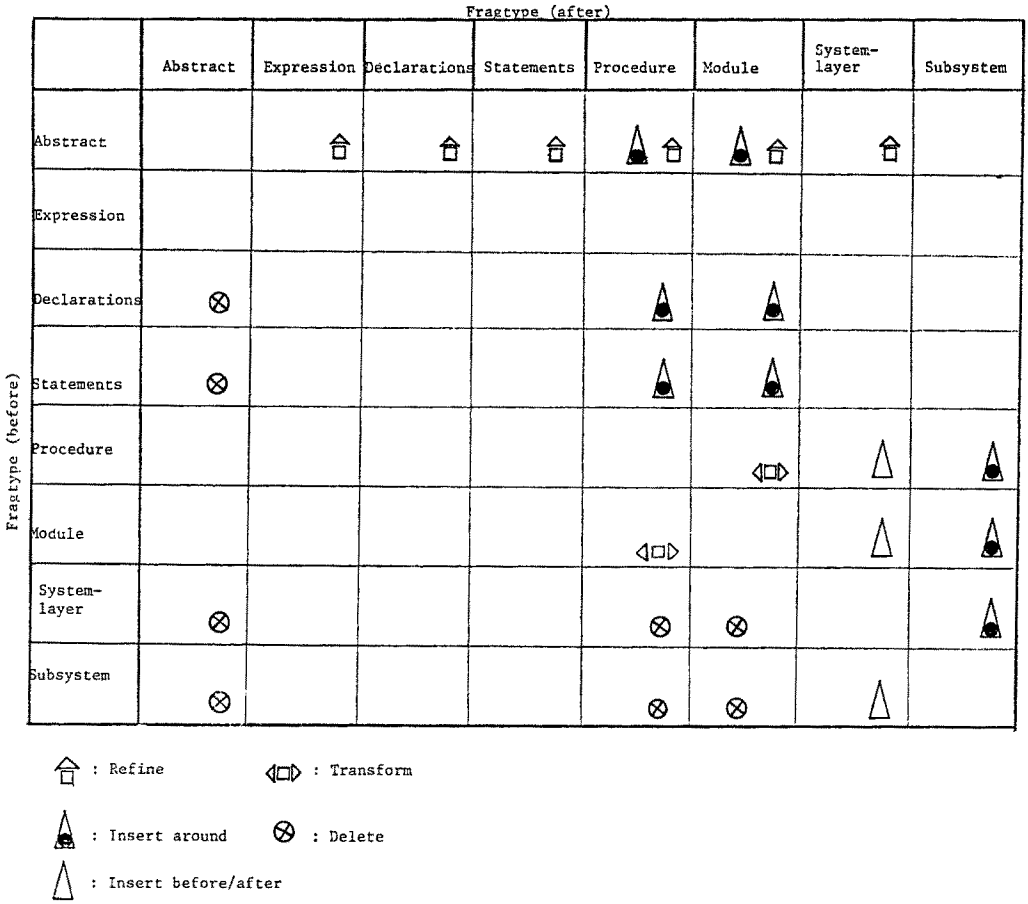

Figure 5 - Operations that trigger off a fragtype change

The *basic form* of the actions that change one fragtype into another are precisely those which are available on the procedure screen and the module screen. In fact, one uses the same editor on the scratch pad, and thus, uniformity is maintained by the system.

3.4. Repository for Building Blocks

The scratch pad derives its power from the formal concepts introduced thus far and the tools that support these concepts. One such tool is the fragment library (FRAGLIB).

FRAGLIB saves, and makes available, fragments of various fragtypes. These fragments are

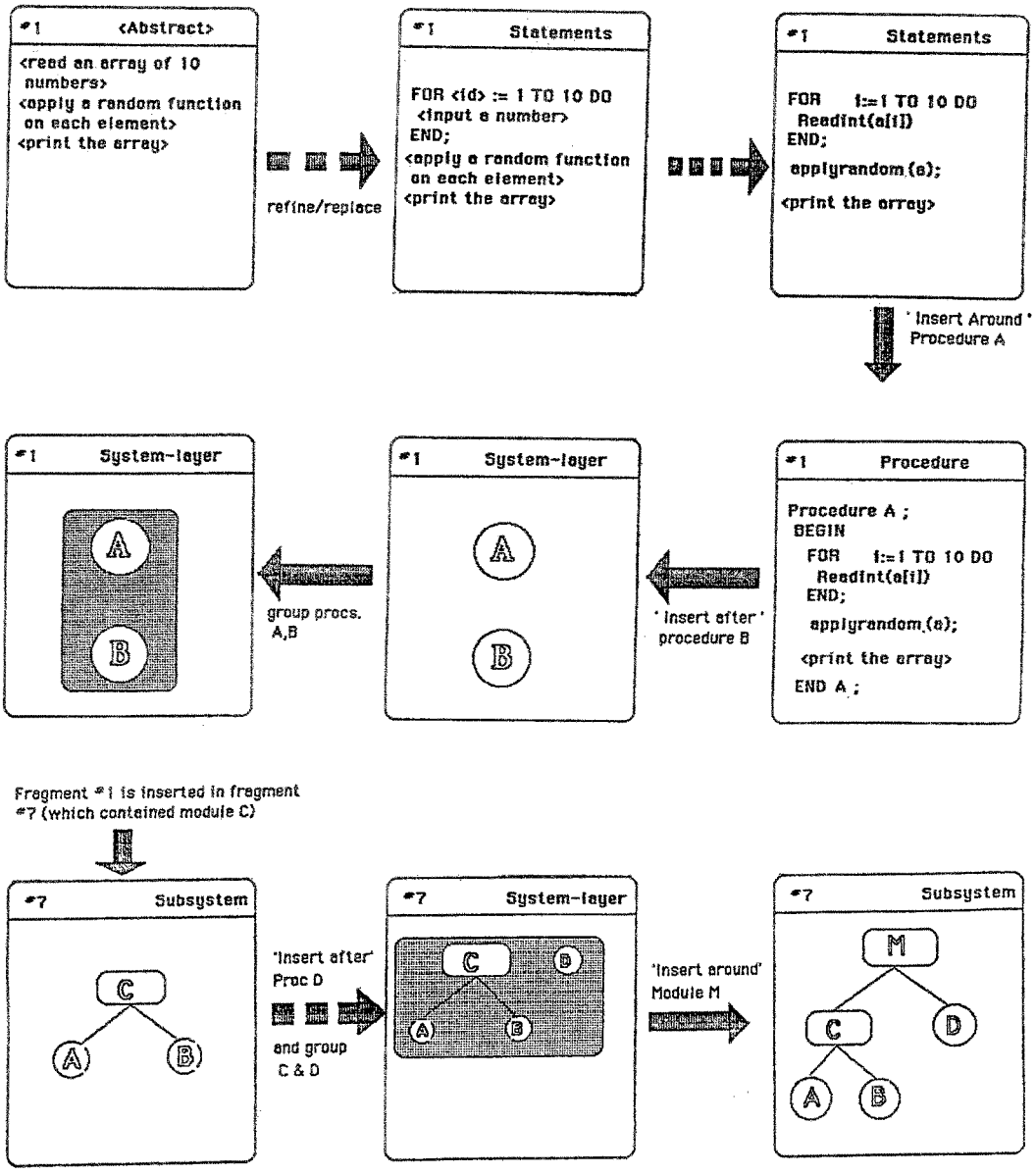

Figure 8 - An example of integrated operations

normally commonly used routines, data structures and algorithms; intra-program usable data structures and algorithmic fragments, and partially completed new fragments, system-layers and sub-systems. FRAGLIB, therefore, is a repository for both complete and incomplete fragments and sub-systems. Together with the other tools provided in the scratch pad, such a facility permits one to rapidly construct prototype, partial or complete systems, as they may not need building from scratch.

The library structure is basically a hierarchy of rings. Each ring holds fragments of various fragtypes. The internal representation of each fragment is the same as those in the scratch pad itself, and those on the other two screens. Thus transporting fragments can be somewhat simplified. In addition, while the current design has no provision for version control of a fragment, such a facility may be included later on top of the kernel library structure.

Parallel work to the idea of a fragment library can be found in TI [Balze81], PSI [Brots81] and PA [Water82]. These three, however, are knowledge-based approaches, which rely on programming clichés, and deal with programming-in-the-small. The last one, in particular, represents program structures as plans, and it provides an editor which operates on such plans.

In MUPE-2, a fragment may be referenced to, from any of the three screens, by its system allocated number or its user given name or description if any. By default, the fragment 'hung' on the current hook is accessed. In addition, a descriptive search facility (such as 'man -k' on UNIX) provides a list of fragments that might be of interest. It is clear that FRAGLIB forms an important and an unavoidable bridge for transporting fragments among the three screens. Without it, the power of MUPE-2 would be severely curtailed.

3.5. Development Tool

Underlying the concept of a fragtype is a single fragtype driven editor which handles both programming in-the-large and in-the-small, and top-down and bottom-up methodologies. This same editor is available in varying strengths on the three screens.

For example, on the procedure screen, the editor will function only on one procedure or a

module at a time (i.e. Procedure or Module fragtype). On the module screen, it will function on the module tree skeleton (i.e. Subsystem fragtype). Yet in the scratch pad, it will vary according to the fragtype of the fragment being edited.

Fragtype changes in the scratch pad (see Figure 5) trigger off dynamic changes in the editing capabilities. This implies that, the editor is context-sensitive [MadVL84]. Thus, when editing a fragment of fragtype Statements, only those features of the editor are active that permit syntactically correct construction of the statements. In addition, by, say, inserting a procedure template around all the statements, the editing capabilities now automatically switch to that which are valid for a whole procedure. Based on the same principle, when the fragtype of a fragment changes from Procedure into System-layer for example, the editing capabilities change from programming-in-the-small to programming-in-the-large.

The uniformity in this *all in one* structured editor is achieved primarily because of its following two main characteristics:

(i) The editor always operates on a fragment of some fragtype, and
(ii) It integrates programming in-the-large and in-the-small, and top-down and bottom-up methodologies.

To the user, this approach results in the following three principal benefits:

(i) The notion of a software building block is concrete.
(ii) The building blocks can be used to construct a well-formed software edifice, and
(iii) The engineering process is versatile.

4. Conclusion

A novel approach to programming is proposed in this paper, to overcome some of the difficulties apparent in programming environments, such as those mentioned in [TeiRe81, DelMS84, FJMPS84] and others. The authors believe that for engineering non-trivial piece of software in an integrated manner, a programming environment should be more than just a structured editor and a run-time system with debugging aids.

In particular, a scratch pad facility which is based on the concept of a fragtype, together with its fragment library, would achieve for reasonably large programs what Pascal has achieved for small programs. That is, formalisation of a fragment and flexibility in its utilisation.

The work described in here is ongoing, but an area of immediate concern is the testing of fragments in the scratch pad. This problem is being approached in two ways. One is the system generated environment for a fragment, and another is a user *hard-wired* environment. While both schemes may be desirable, the latter appears to be a non-trivial task for dynamic data structures [MadWi81, Madha84].

Finally, MUPE-2 owes much to the recent and current research in programming environments, which has pointed out the need for a scratch pad facility.

ACKNOWLEDGEMENT

The work described in this paper was in part supported by FCAC, Quebec, Canada under Grant 290-19.

5. References

[Alfor81] Alford, M.W.: *SDS: Experience with the Software Development System.* In Software Engineering Environments, (ed) Hünke, H., North Holland Pub. Co., Amsterdam, 1981.

[ArcCo81] Archer, J.E., Conway, Jr and R.: *COPE: A Cooperative Programming Environment.* Technical Report 81-459, Cornell University, June 1981.

[Balze81] Balzer, R.: *Transformational Implementation: An Example.* IEEE Trans. Soft. Eng., Vol. SE-7, Jan. 1981, pp. 3-14.

[Brots81] Brotsky, D.C.: *Program understanding through cliché recognition.* M.S. thesis proposal, MIT, Cambridge, MA., 1981.

[DHKL84] Donzeau-Gouge, V., Houet, G., Kahn, G., Lang, B.: *Programming Environments Based on Structured Editors: The MENTOR Experience.* In Interactive Programming Environments (eds.) Barstow,D.R., et al., McGraw-Hill, 1984.

[DelMS84] Delisle, N.M., Menicosy, D.E., Schwartz, M.D.: *Viewing a Programming Environment as a Single Tool.* Proc. ACM SIGSOFT/SIGPLAN Soft. Eng. Symposium on Practical Software Development Environments, ACM Sigplan Notices, Vol. 19, No. 5, May 1984, pp. 49-56.

[FJMPS84] Fischer, C.N., et al.: *The Poe Language-Based Editor Project* Proc. ACM SIGSOFT/SIGPLAN Soft. Eng. Symposium on Practical Software Development Environments, ACM Sigplan Notices, Vol. 19, No. 5, May 1984, pp. 21-29.

[Ivie77] Ivie, E.L.: *The Programmer's Workbench - A machine for Software Development.* Comm. ACM, Vol. 20, No. 10, Oct. 1977, pp. 746-753.

[Madha84] Madhavji, N.H.: *Visibility Aspects of Programmed Dynamic Data Structures.* Comm. ACM, Vol. 27, No. 8, Aug. 1984, pp. 764-776.

[MadVL84] Madhavji, N.H., Voulíouris, D. and Leoutsarakos, N.: *The Importance of Context in an Integrated Programming Environment.* To appear in the Proc. 18th Annual Hawaii Int. Conf. on System Sciences, Hawaii, Jan. 1985.

[MadWi81] Madhavji, N.H., and Wilson, I.R.: *Dynamically Structured Data.* Software-Practice and Experience, Vol. 11, No. 12, Dec. 1981, pp. 1235-1260.

[MedNo81] Medina-Mora, R., Notkin, D.S.: *ALOE users' and implementors' guide.* Tech. Rep. CMU-CS-81-145, Dept. of Comp. Science, Carnegie-Mellon Univ., Pittsburgh, Pa., Nov. 1981.

[Reiss84] Reiss, S.P.: *Graphical Program Development with PECAN Program Development Systems.* Proc. ACM SIGSOFT/SIGPLAN Soft. Eng. Symposium on Practical Software Development Environments, ACM Sigplan Notices, Vol. 19, No. 5, May 1984, pp. 30-41.

[Snowd81] Snowdon, R. A.: *CADES and Software System Development.* In Software Engineering Environments, (ed) Hünke, H., North Holland Pub. Co., Amsterdam, 1981.

[TeiRe81] Teitelbaum, T., Reps, T.: *The Cornell Program Synthesizer: A syntax directed programming environment.* Comm. ACM, Vol. 24, No. 9, Sept. 1981, pp. 563-573.

[Water82] Waters, R. C.: *The Programmer's Apprentice: Knowledge Based Program Editing.* IEEE Trans. Soft. Eng., Vol. SE-8, No. 1, Jan. 1982, pp. 1-12.

[Wilan80] Wilander, J.: *An Interactive Programming System for Pascal.* In Interactive Programming Environments, (eds.) Barstow, D.R., et al., McGraw-Hill, 1984.

[Wirth82] Wirth, N.: *Programming in Modula-2.* Springer Verlag, 1982.