Context-Sensitive Editing in Orm

Görel Hedin

LU-CS-TR:92-108

Also Published in: Proceedings of the Nordic Workshop on Programming Environment Research, Tampere University of Technology, Software Systems Lab. TR 14, Tampere, Finland, 1992

Department of Computer Science
Lund University

P.O. Box 118
S-221 00 Lund
Sweden

Context-Sensitive Editing in Orm

Görel Hedin

Lund University, Department of Computer Science
Box 118, S-221 00 Lund, Sweden.
e-mail: Gorel.Hedin@dna.lth.se

September 1991

Abstract

Context-sensitive editing utilizes static-semantic information such as scope rules, properties of declared names, and context of the current edit focus to support high-level editing. Typed references, as present in statically typed object-oriented languages, make it possible to determine the legal set of messages to an object and provide quick access to these. The Orm programming environment for Simula includes such advanced editing facilities. We give an overview of these facilities and report on the practical experiences of such editing support. We also comment on how object-oriented attribute grammars can be used to implement context-sensitive editing.

1. Introduction

Structure-oriented programming environments with incremental context-sensitive analysis have usually been targeted for novice users. Emphasis has been put on providing support for building syntactically correct programs and giving feedback on compile-time errors during editing. While such facilities can be helpful it has been argued that they are not so important to the experienced programmer who usually knows the syntax very well. However, by making use of the context-sensitive information kept up to date by the incremental analyzer, these environments have a large potential for giving advanced editing and browsing support which can be very helpful also to the experienced programmer.

Orm [Mag90] is an incrementally compiling programming environment for Simula [DMN68]. It includes an incremental static-semantic analyzer which incrementally maintains context-sensitive information after each single edit step. Currently, three editing mechanisms are available in Orm: menu-driven structure-oriented editing, textual input which is incrementally parsed, and context-sensitive editing which will be further explained in this paper. These different editing mechanisms can be freely mixed during the editing of a program.

2. Context-sensitive editing in Orm

2.1 The Names menu

Context-sensitive editing in Orm is centered around the *Names* menu which provides a list of all declared names visible at the current edit focus. For example, when the current focus is a normal expression in a procedure, the Names menu shows the declared names in that procedure, in the surrounding class, in superclasses of the surrounding class, etc.

A *remote access* expression consists of an object-expression and a selector-expression separated by a dot, e.g.

```
Arthur.turnRight
```

Here, `Arthur` is a reference variable denoting an object of, say, class `Turtle`, and `turnRight` is a procedure remotely accessed via the `Arthur` reference variable. If the current edit focus is the selector-expression of a remote access, the Names menu presents all names declared in the class (and superclasses) of the remotely accessed object. Thus, if `turnRight` is the current focus, the Names menu presents all the names declared in class `Turtle` and its superclasses.

Selecting an entry in the Names menu causes the current edit focus to be replaced.

2.2 Selecting procedures, arrays, and classes

Some of the entries in the Names menu denote procedures, arrays, and classes. Normally, when a procedure entry is selected, the current edit focus is replaced by a procedure call with placeholders for the appropriate number of actual parameters. However, depending on the local context, this is not always appropriate. For example, the result of a function `p` is returned by assigning a value to `p`, thus treating the procedure name as a variable. Therefore, when the current edit focus is the left side of an assignment inside the function `p`, selecting `p` in the Names menu is not interpreted as a call, but results only in insertion of the identifier `p`.

Arrays and classes similarly may have different interpretations depending on the local context, and placeholders for array indices and class parameters are inserted as appropriate. For classes, the keyword "`new`" is also inserted if this is appropriate in the context.

2.3 Creating remote accesses

Many of the entries in the Names menu denote reference variables or procedures which return reference values. For these entries, there is a pull-right submenu which includes the names declared in the class (and superclasses) of the reference.

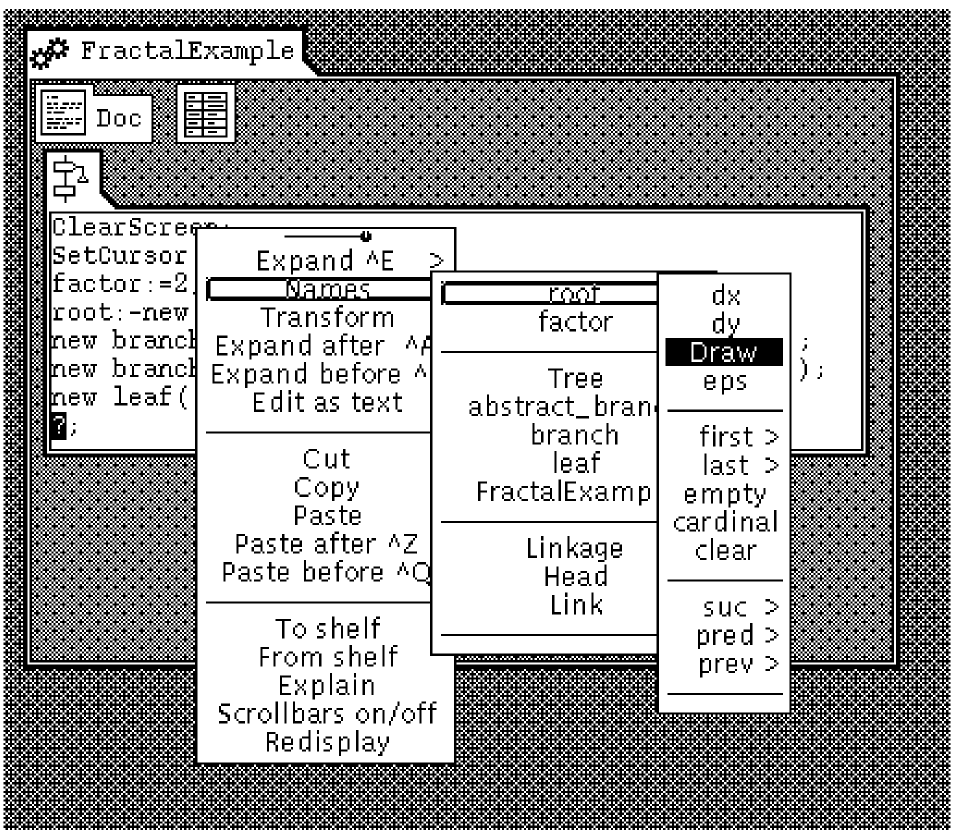
This mechanism works recursively for the entries of the submenu. Selecting an entry in a submenu results in the current edit focus being replaced by a remote access.

2.4 Using names which are not declared

If a name is used which does not have a declaration, this name will turn up in the Names menu of the block (procedure or class) where it is used, marked as "undeclared". Thus, even if a name is not declared, it can be retrieved from the Names menu as soon as it has come into use. If the user decides to declare the name, it can also be fetched from the Names menu. This gives the effect that a name only has to be typed once.

2.5 An example

The snapshot below shows an example of using the Names menu in Orm.



The figure shows context-sensitive editing in the body of a procedure `FractalExample`. The edit focus is the last unexpanded statement in the procedure, indicated with a "`?`" placeholder in inverse video.

The leftmost menu is the standard popup edit menu which can be pinned up for convenient access. In the figure, the user has pulled right over the Names command, bringing up the first submenu starting with `root` (which is a reference variable), and further pulled right over `root`, bringing up the second submenu starting with `dx`, and is currently holding the mouse over the `Draw` entry in the this submenu. Releasing the mouse button at this point would cause the edit focus to be replaced by `root.Draw( ? , ? , ? )` where the question marks are placeholders for the actual parameters of the `Draw` procedure.

The first submenu shows the names visible at the edit focus. The submenu is divided into sections, one for each visible scope. The top section shows the names declared locally in the procedure (`root` / `factor`). The middle section shows the names declared in the program (`Tree` / `abstract_branch` / `branch` / `leaf` / `FractalExample`). The lower section shows the names declared in an imported module (`Linkage` / `Head` / `Link`).

The name `root` is a reference variable of class `Tree`. The right-most submenu therefore shows all the names accessible in `Tree` objects. The class `Tree` is a subclass of `Link` which is in turn a subclass of `Linkage`. The three sections of the submenu shows the names declared in these three classes. The top section shows the variables and procedures declared in `Tree` (`dx` / `dy` / `Draw` / `eps`). The middle section shows the procedures declared in class `Link` (`first` / `last` / `empty` / `cardinal` / `clear`). The third section shows the procedures declared in class `Linkage` (`suc` / `pred` / `prev`).

Several of the names in the right-most submenu are functions which return a reference value. These are indicated with a "`>`" mark in the menu (e.g. "`first >`" and "`last >`"). At these entries it is possible to pull right to bring up yet another submenu. In this way, arbitrarily long remote-access expressions can be constructed.

2.6 Type filtering

In the current implementation of Orm, the Names menu shows all names in the declarative environment, regardless of if inserting them at the current edit focus would cause type-checking errors or not. For example, placeholder "`?`" in the following expression:

```
a + ?
```

must obviously be replaced by an integer-valued expression in order to not give a type-checking error. However, the Names menu will at this point show all visible names: class names, procedure names, and variable names of all types. An obvious extension would be to compare the types of the declared names with the expected type of the current edit focus. Names of non-conforming types could either be removed from the menu or presented in a different manner . One way of presenting the names could be to use white background for type-conforming entries, light-grey for non-conforming but "pull-right-able" entries, and darker grey for non-conforming non-reference-valued declarations.

Keeping also non-conforming entries in the menu would have the advantage of keeping the same menu structure throughout a procedure. Such regularity can be very important from an interaction point of view since users adapt very quickly to such structures and start "mousing ahead".

3. Attribute-guided implementation

The incremental static-semantic analyzer in Orm is implemented using a variant of attribute grammars which allows object-structures to be built and maintained by the attribute system [Hed88]. The attribute grammar is written in an object-oriented notation [Hed89], allowing syntax nodes to be classified in several abstraction layers, from very general abstractions, e.g. `Node`, to intermediate, e.g. `Expression`, to very specialized, e.g. `Addition`. The object-oriented notation also allows "demand" attributes which are implemented as virtual functions. Demand attributes are useful in incremental environments because they can be specified just as normal attributes but consume no space. They are thus suitable for attributes which are either very simple to compute or accessed comparatively infrequently , e.g. by facilities in the user interface. This attribute grammar approach turned out to be a suitable base for implementation of the context-sensitive editing functionality .

In our attribute grammar for Simula, each node in the syntax tree has a demand attribute `env` which is a reference to an object structure, representing the declarative environment of the node. The `env` attribute is declared in the most general syntax node class and is therefore accessible in the same way for all nodes. When the Names menu is activated, the context-sensitive editor simply accesses the `env` attribute of the node and traverses the object-structure referenced by this attribute to build up the entries in the Names menu. This works for all syntax nodes: e.g., identifiers, expressions, statements, and declarations.

As mentioned in section 2.2, the interpretation of a name depends on the local context of the edit focus. This interpretation can also be handled by using attributes. E.g., to determine if a procedure name should be interpreted as a call or as a return variable, let us define a boolean demand attribute `ReturnVariablePosition` which returns `true` for those expressions which could be replaced by a procedure name acting as a return variable. This is for instance the case for the left son expression of an assignment. If we assume that the (object-oriented) context-free syntax of node classes `Expr`, `Node`, and `Assignment` is already defined elsewhere, the `ReturnVariablePosition` attribute can be specified as an extension to these classes as follows:

```
Expr
{  inh ReturnVariablePosition: boolean demand;    (* 1 *)
};

Node
{  son Expr.ReturnVariablePosition := false;      (* 2 *)
};

Assignment
{  left.ReturnVariablePosition := true;           (* 3 *)
}
```

(1) is a declaration of the attribute `ReturnVariablePosition` as an inherited demand attribute of all `Expr` nodes. (2) is an equation saying that the attribute `ReturnVariablePosition` of all `Expr` son nodes has the value `false`. (3) is an equation in the `Assignment` node class which overrides the equation (2) for its left son node.

In the current implementation of context-sensitive editing we have only used whatever attributes were already available (for the incremental static-semantic analysis). However, we plan to try a more systematic attribute-based approach using attributes specifically designed for use by the context-sensitive editor. In addition, the desired functionality of the context-sensitive editor should be specified in a grammar-like description, rather than being explicitly coded as it is now.

To implement type filtering, the attribute grammar approach also seems very promising. Possibly, it could be useful to extend the attribute grammar formalism with functions with parameters. E.g., a boolean function `TypeAcceptable(T)` could be defined for all expressions, returning `true` if replacing the expression by another expression of type `T` would not give a type-checking error. The definition of `TypeAcceptable` would benefit from using the attributes in a similar way as is already done for type checking in the static-semantic analyzer.

4. Experience of usage

The context-sensitive editing facility in Orm has been extremely successful. Some of the advantages over textual input of identifiers are the following.

- *Quicker*. It is usually much faster to pick a name from the Names menu than to type it on the keyboard.
- *Memorizing names unnecessary*. The many names in a program are hard to keep track of and more so the larger the program is, and even more so when using modules written by somebody else. The Names menu makes it unnecessary to memorize the exact wording and spelling used.
- *No mouse/keyboard conflict*. Orm is an extremely interactive programming environment based on direct manipulation, where the mouse is the main input device. With the context-sensitive editing facilities it is usually necessary to

type a name only once on the keyboard. From then on it can be picked from the Names menu and the user does not have to shift back and forth between mouse and keyboard.

- *Browsing*. The Names menu is often used as a simple browsing facility, to see what classes and procedures are available.

The context-sensitive editing in Orm is a general-purpose mechanism which could be adapted to any language. However, it suits object-oriented languages particularly well. E.g., when editing procedures of a class, the instance variables declared in the class are directly visible. Also, the use of statically typed references and remote access makes it possible to determine the legal set of messages (procedure calls) to an object and provide quick access to these. Using Names on a procedural language would usually result in a very long list of names on the outermost level. Modules could be used for structuring the menus, but not in such a semantically meaningful way as is done by the remote access mechanism in object-oriented languages.

As an experiment, Orm was used in an introductory programming course for first-year undergraduates at Lund Institute of Technology. A group of 15 students used Orm. The students were of mixed background; most of them had done some programming before (in high school or at home), but there were also some who had never programmed before. However, all students were novices of Simula. Each student used Orm at 18 lab-exercises and for 3 individual small programming projects. At the end of the course a questionnaire of 73 questions was issued which was answered by all 15 students. Two of the questions related to the context-sensitive editing facility and were answered as follows:

Q: How did you like the “Names” facility?

A:

Very good	100 %
Good	0 %
Not so good	0 %
Bad	0 %
Very bad	0 %
No answer	0 %

Q: Did you use “Names” or “Edit as text” most often at input of variables and procedure calls?

A:

Almost only “Names”	67 %
More “Names” than “Edit as Text”	27 %
Both equally often	7 %
More “Edit as Text” than “Names”	0 %
Almost only “Edit as Text”	0 %
No answer	0 %

Although these questions are rather superficial and the student group rather small, there is no doubt about the positive response on the “Names” facility.

5. Conclusion

Context-sensitive editing is an example of the kind of advanced support which is possible to implement in programming environments based on fine-grained incremental compilation techniques. The Names mechanism reported in this paper is a very effective editing technique, yet simple to use. The mechanism is also simple to implement, once the static-semantic information maintained by the incremental compiler is available. The present mechanism used in Orm is only a first step towards more effective and supportive programming environments, useful not only for the novice programmer having trouble with syntax, but useful also for the experienced programmer whose main problems are in keeping track of a large programming system and its internal dependencies. Future research in this area includes further development of advanced editing and browsing support and also development of grammar-based techniques for implementing such support.

Acknowledgments

This research was carried out within the Mjølner projects, partially financed by the Nordic Fund for Technology and Industrial Development (Nordisk Industrifond) and by the Swedish National Board for Technical Development (STU).

References

[DMN68] Dahl O.-J., Myhrhaug B., Nygaard K. SIMULA 67 Common Base Language. Norwegian Computing Centre, Oslo, 1968. Revised 1970, 1972, and 1984. Swedish standard SS 63 61 14, 1987.

[Hed88] Hedin G. Incremental Attribute Evaluation with Side-effects. Proceedings of the Workshop on Compiler Compilers and High Speed Compilation. Berlin, October 1988. Springer-Verlag. LNCS 371. 175-189.

[Hed89] Hedin G. An Object-Oriented Notation for Attribute Grammars. Proc. of the European Conference on Object-Oriented Programming, Nottingham. Cambridge University Press, 1989, 329-345.

[Mag90] Magnusson B., et al. An Overview of the Mjølner/Orm Environment: Incremental Language and Software Development. Proc. of the 2nd Int. Conf. on Technology of Object-Oriented Languages and Systems. Paris 1990, 635-646.