An Overview of the Mjølner Orm Environment:
Incremental Language and Software Development

Boris Magnusson, Mats Bengtsson, Lars-Ove Dahlin,
Göran Fries, Anders Gustavsson, Görel Hedin, Sten Minör,
Dan Oscarsson, and Magnus Taube

LU-CS-TR:90-57

Also published in *Proceedings of the 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, 1990

Department of Computer Science
Lund University

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

An Overview of The Mjølner/ORM Environment: Incremental Language and Software Development

Boris Magnusson, Mats Bengtsson, Lars-Ove Dahlin, Göran Fries, Anders Gustavsson†, Görel Hedin, Sten Minör, Dan Oscarsson, Magnus Taube

Department of Computer Science
Lund University
P.O.Box 118
S-221 00 Lund
Sweden

†TeleLogic AB
Malmö, Sweden

Abstract

This is an overview of the incremental language and software development environment ORM, which is primarily designed to support programming in object oriented languages. Its characteristics are a consistent user interface built on hierarchical windows and a consistent approach to defining language dependent parts. The paper presents the user view of the system which is based on graphical representation and a direct manipulation interface. This interface is used throughout the system: in program development, in program execution observation, and when developing grammars. The language dependent functionality is specified in grammars which are interpreted by the environment. Both grammars and programs are stored in the same structured format. The program execution is based on incremental compilation. In this paper we give an overview of the motives behind the project, design issues and major implementation techniques used. ORM has been developed as a part of the Nordic Mjølner project.

1. Introduction

In the development of industrial high quality software we regard it as essential to provide languages that support early error detection, such as static typing. Turn around time in systems for such, usually compiled, languages is, however, a major problem. This is in contrast to interactive systems based on interpretation [Sandewall 78]. These environments have often shown an attractive, at least short term, decrease in development time. They do, however, not support early error checking. In the ORM system we have set out to provide both: supporting strongly typed languages with an incremental compiling environment. ORM provides compile time error feedback after each edit operation as well as immediate execution after a program change.

Another problem in traditional so called "tool based" environments is the proliferation of tools, all with different interfaces. In the ORM environment we have designed a direct manipulation graphical user interface that is consistently used in program manipulation, observation of program execution, and when defining program grammars. In the ORM environment the functional components are integrated into one single tool and use a common program representation. The representation is built on program structures rather than source text. This makes it possible to integrate additional functionality into the system based on a structured view of programs. This is essential since we have certainly not yet developed all the functionality we would like to see in a full blown industrial strength programming environment.

The ORM environment is generic and can be specialized to support different languages. The technique used for specialization is direct interpretation of grammars defining the actual programming languages. Currently there are grammars for: abstract and concrete syntax, semantic checking and code generation. The grammars have been developed with the use of ORM itself which has shown to us the value of interactive program development also in language implementation. A change to one of the grammars takes effect immediately. The use of grammar interpretation also makes the system flexible and easy to customize by adding or removing language constructs, support special syntax, or even use ORM as a language laboratory.

This paper focuses on the major contributions of the ORM environment:

•Consistent use of hierarchical windows to represent tree-structures: nested program constructions, runtime data structures, filesystem directories and grammar aspects

•Direct manipulation of objects in an integrated environment rather than interaction with tools

•Use of grammar interpretation for providing language dependent support

•Structuring of grammars after different aspects: abstract and concrete syntax as well as static and dynamic semantics

•Interactive development of language grammars with immediate feedback

•Incremental semantic analysis after each edit operation

The work reported in this paper has been carried out as part of the Nordic Mjølner project [Dahle et al. 87]. Our work is supported by The Nordic Fund for Industrial Development and The Swedish Fund for Technical Development (STU). The Mjølner project has a long range goal of increasing the efficiency in the software development process in general. This problem has been attacked in different ways in different sub-projects: In the ORM environment the major emphasis has been on making the computer a better and more efficient tool in the programming process. As presented in this paper ORM supports programming in SIMULA. Grammars for a part of BETA [Kristensen et al. 87] have also been developed. The design of ORM is general and future activities will be directed towards supporting also other languages. Other parts of the Mjølner project has focused on other aspects: simple but more expressive languages (BETA) and methodology and support for specification (OSDL [Møller-Pedersen et al. 87]). Common interests in the project have been use of object oriented technology and focus on systems supporting large, or industrial, software.

The rest of this paper is organized as follows. In section 2, the user interface based on direct manipulation and visualization of contexts is described. In section 3, the use of ORM as a programming environment is presented. Section 4 explains how a language is implemented by editing grammar objects. Section 5 contains a summary of major design issues in ORM. Finally, in section 6, the implementation is described.

2. The User Interface

The ORM user interface is based on two major principles: (1) direct manipulation and (2) visualization of context by means of a hierarchical window system. These principles are applied consistently throughout the system to present the objects of major interest in the system: programs, program executions, and grammars. The system is open-ended and allows also ordinary UNIX files and directories to be presented in the same way.

Objects of major interest, e.g. programs and their internal parts like procedures and classes, are represented as windows on the screen. The user can *identify* a window with the object it represents. This identification is a basic prerequisite for direct manipulation. Each window has its own set of operations which are relevant to the object and operations such as moving, cutting, and pasting can be performed directly.

In a direct manipulation system the physical object structure is of great importance. An object always resides at a particular location, which is usually vital for its meaning. E.g. the meaning of a procedure (method) "draw" depends entirely on which class it resides in. For some kinds of objects, the location is more of organizational than of semantic importance, e.g. the user organizes files inside folders. This physical structure is directly visualized in ORM by means of a generalized window system. In traditional window systems the windows reside on the screen and can overlap each other and be iconized to preserve screen space. In the ORM window system this principle is generalized so that not only the screen can contain windows, but *any* window can contain a local set of overlapping iconizable windows. This gives a hierarchical organization of windows which is employed to match the physical object structure. E.g., a class which contains two procedures: "draw" and "scale" is visualized by a class window which contains two procedure windows. Moving the "draw" window to another class window is an editing operation on the program. The context of a particular object is always clear due to the hierarchical organization of windows.

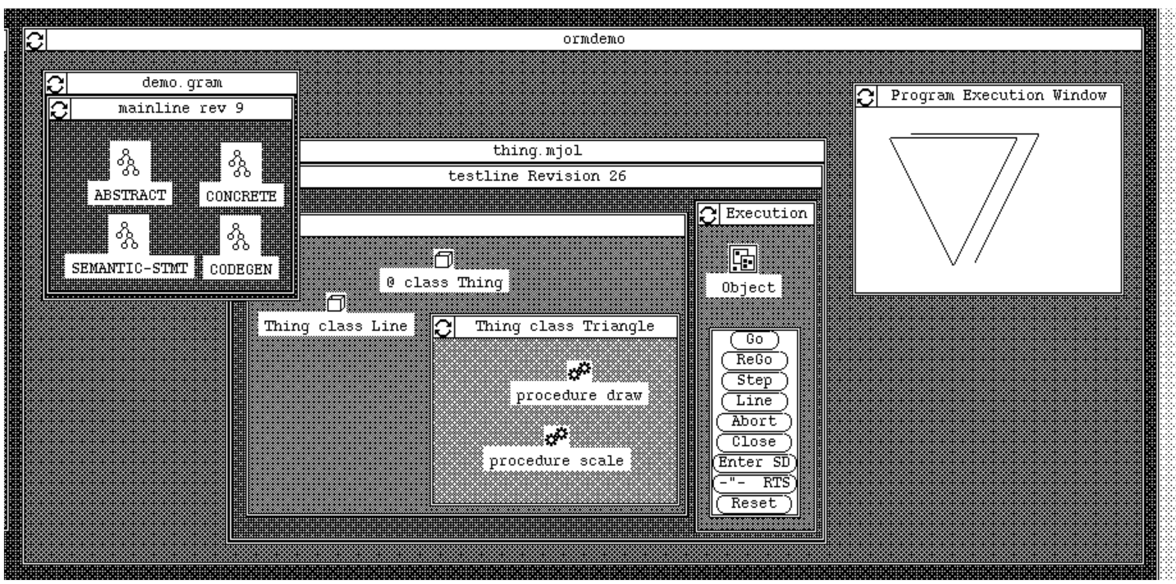

Figure 1. Snapshot from an ORM session

In the snapshot from an ORM session shown in figure 1, the outermost window represents a UNIX directory ("ormdemo") containing a grammar object ("demo.gram"), a program object ("thing.mjol") and the I/O window for an executing program ("Program Execution Window"). The program window contains the source code with procedures nested inside classes. Different shades of grey in the window backgrounds enhance the hierarchical window structure.

The hierarchical window system provides a basic navigation aid in terms of the physical object structure: an object can be located by opening its surrounding windows. The status of inner windows (e.g. icon/window state, placement within the outer window, and size) is preserved as a window is iconized, moved, etc. E.g. when iconizing the program window, all its interior parts (main block, classes and procedures) will disappear from view, but reappear in exactly the same state when the program window is opened again.

In addition to the structural navigation supported by the window system, there are many important relations that the user may want to navigate along. In particular, there are many such relations in programs, e.g. super/subclass relations between classes and declare/use relations between identifiers in a program, just to name a few. Such relations are handled by means of *hyper linking*. When the user clicks on a link button, the target window is made visible (opening outer windows if needed) and visual feedback is given on the target window by flashing its contents. Thus, the object structure can be seen as basically hierarchical but with possibilities to follow links which then quickly bring another part of the hierarchical structure into view.

The direct manipulation interface of ORM is similar in flavor to the Star office automation system [Johnson et al. 89] and to successors of these ideas, e.g. the Macintosh finder. The ORM interface differs from STAR and Macintosh by the introduction of hierarchical windows which always shows an object (window) in its context. We think this is a particularly valuable property in programming environments for object oriented languages where the context has strong semantic importance. Earlier programming environments which utilize graphics and windows usually follow the tool-based approach where the windows represent tools rather than objects, e.g. the Smalltalk environment [Goldberg 84] with its browser and inspector tools. The ideas on user interface used in ORM are further developed in [Taube et al. 87] and [Hedin et al. 88].

3. Programming in ORM

The main objects of interest to a programmer are programs and program executions. A program is internally composed of blocks, e.g. classes and procedures. A program execution is an incarnation of a program and is composed of block instances, e.g. class instances (objects) and procedure activations. Documentation is viewed as part of the program rather than as a separate document.

Figure 2 shows an example of editing and execution of a program object. The source program contains classes modelling graphical "things" and the execution window contains instances of these classes. The program execution is half way through drawing some of the "things" and the output drawing appears in the I/O window.

The goal has been to allow the user to focus on the program and execution objects, rather than tools. Thus, compilation activities, such as semantic checking, code generation, linking, and loading, are all done automatically as needed by the system without need for initiation by the user.

Source program

Program blocks, represented as windows and icons, can freely be laid out in 2-dimensions. Each block contains its "locals" (i.e. parameters and variable declarations) and its "body" (statements). These are presented as a textual unparsing of syntax trees and are edited in a structure-oriented way. The usual (and in our opinion unnatural) view of a program as a linear text has thus been abandoned.

Static semantic checking is performed incrementally as the user edits the program and continuous monitoring of static semantic errors is provided. Errors are marked unobtrusively and can be corrected at any time. An explanation of the error can be obtained via a menu command. Both erroneous and incomplete programs can be executed, giving the effect of a breakpoint at the error.

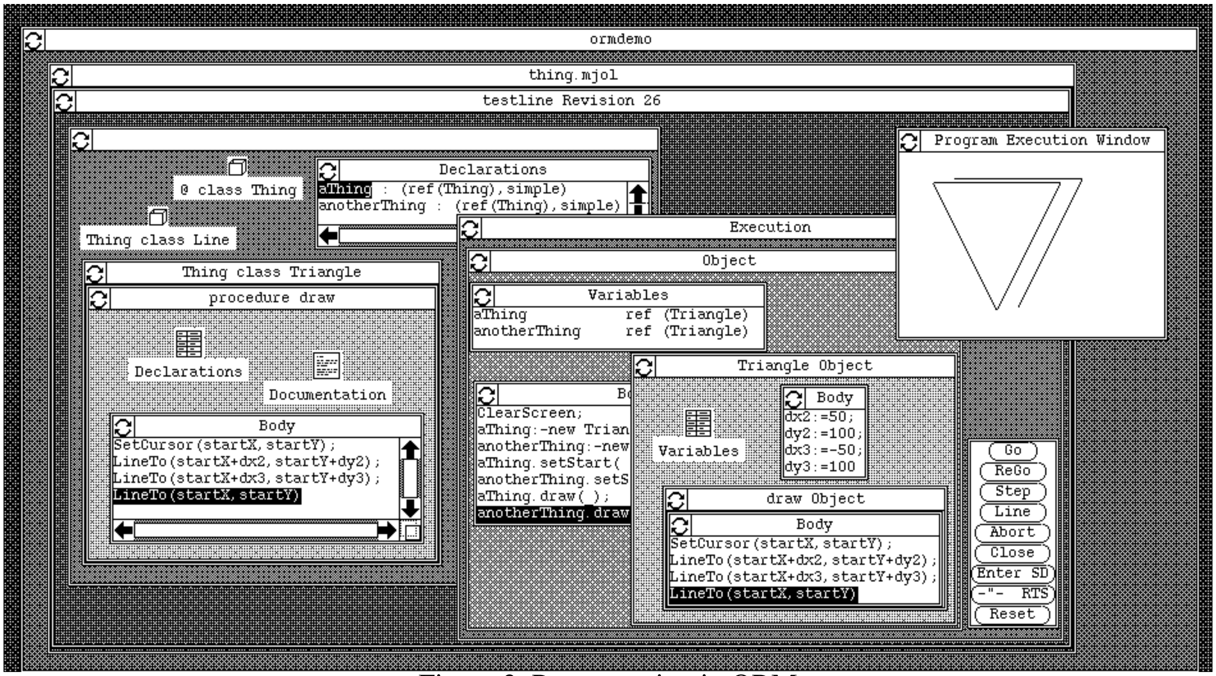

Figure 2. Programming in ORM

Program execution

In the execution window, objects and procedure activations are shown nested according to the block structure. The present state shows an activation of the draw procedure of a Triangle object. The draw activation belongs to the Triangle object and is therefore located inside it. Inside the draw activation its locals and body windows are shown. These windows are essentially instances of their templates in the program source, but the locals window contains the values of the variables rather than their types as in the source. The code window shows the current (re-)activation point of the draw activation. For reference (pointer) variables, the value field contains the actual qualification of the referenced object, and works like a link button which links to the referenced object. E.g. if the user clicks on the variable "anotherThing" the "Triangle" object's window will flash. For simple variables, such as integers and booleans, the value is shown directly in the locals window.

The link facility for reference variables provides a way of navigating in object structures. The variable "aThing" references a Triangle object other than the one currently shown. This object will appear if the user clicks on the variable. Another way to bring up windows is via the call stack (not shown in the example). Only objects which the user has followed links to appear in the execution window, and it is possible for the user to remove those object windows which are no longer of interest. The reference variables act as starting points for exploration by the user.

Incremental Execution

In a direct manipulation environment such as ORM it is natural to change programs while they are executing. E.g. new procedures can be added to classes of which objects already exist, behavior of procedures can be changed, and so on. Such functionality can be very useful when developing programs in the exploratory fashion, and it can be a necessity when maintaining persistently executing programs. This kind of functionality gives rise to consistency and version issues and also to synchronization issues in case of on-the-fly updates to a running program. At present, ORM does not yet contain this functionality, but experiments have been made in an earlier prototype environment and the consistency and version issues have been addressed in [Hedin et al. 86].

4.Implementing a language in ORM

ORM differs from most other structure-oriented environments by its grammar interpretive approach. Several other systems, such as Mentor [Donzeau-Gouge et al. 80], Gandalf [Notkin 85], The Synthesizer Generator [Reps et al. 84], and Centaur [Borras et al. 88] supports *generation* of language specific environments from language grammars whereas ORM *is* a generic environment that is parameterized with language specific information by interpreting language grammars dynamically. Other environments based on interpretation of language descriptions are DOSE [Feiler et al. 86] and Muir [Winograd 87]. A consequence of the grammar interpretative approach is that the environment need not be re-generated in order to handle various languages or to adapt to a modification in the language specification. Interactive modifications of grammars are supported explicitly in the ORM environment. When a grammar has been modified the changes come into effect in the program depending on the grammar instantly. Thus ORM supports interactive modifications and construction of both programs and *languages* within the environment. This facility has been used within the project for developing grammars for the programming languages Simula [SIS 87], and parts of BETA [Kristensen et al. 87]. It has also been used extensively for developing formalisms for grammars used within ORM itself expressed by means of metagrammars as described below.

The previous section showed the editing and execution of a program object within ORM. Grammars are manipulated by opening a grammar object that is connected to a program object. Editing a grammar object includes developing the syntax and static as well as dynamic semantics. A grammar object in ORM has, just like a program object, an internal structure. It consists of four *aspects* of a language each specified in a separate formalism. The division of the aspects into different grammars in different formalisms is a result of the aim to make ORM open-ended. The possibility to add new language aspects if necessary and to make configurations of ORM where some aspects are not present is then vital. Furthermore it is possible to experiment with different formalisms for a certain language aspect without affecting the other aspects.

•The *abstract grammar* defines the structure of the language. It is specified in a structured variant of extended BNF in which all concrete syntax has been removed.

•The *concrete grammar* defines the textual presentation for the language constructs defined in the abstract grammar. This includes specification of the syntactical sugar and presentation layout for each language construct.

•The *semantic grammar* defines the static semantics, such as scope-rules, identifier bindings, and type checking. It is expressed in an object-oriented variant of attribute grammars, allowing attributes and equations to be inherited between productions. This allows behavior to be specified at the appropriate level of generalization [Hedin 89]. Furthermore they allow side-effects in a controlled manner which facilitate efficient incremental attribute evaluation [Hedin 88].

•The *code-generation grammar* translates the language constructs into an intermediate language that expresses the dynamic semantics of the language. Also in this case an object-oriented variant of attribute grammars is used.

Figure 3 shows editing of the grammar object "demo.gram". The four aspects constituting the internal structure of the object have all been opened and may thus be manipulated by means of structure-oriented editing. The fact that the grammar is interpreted results in that the various grammars may be modified incrementally and that each change comes into effect instantly. For instance, adding a new production in the respective grammar automatically results in that the connected program object may be manipulated and executed according to the new version directly, i.e. without escaping the environment or making any re-generations of the environment.

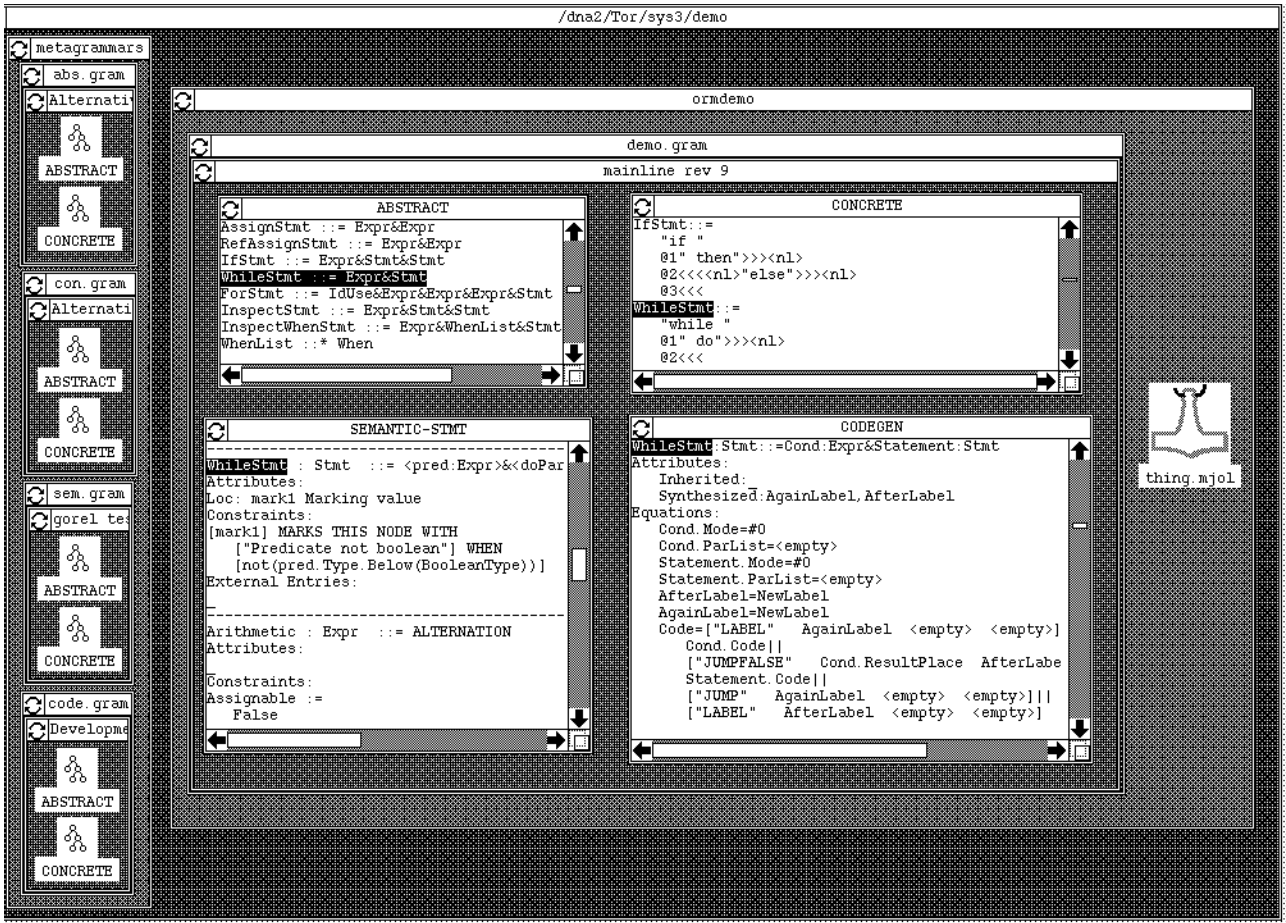

Figure 3. Editing a grammar object

Grammar objects are edited using the same structure-oriented editor as for program objects. The editor interprets metagrammars when editing these language grammars. E.g. the metagrammar, "abs.gram", in figure 3 describes different aspects of abstract grammars. Similarly, the other language grammars are described by a corresponding metagrammar. Some of the metagrammars are self-descriptive. E.g. the abstract grammar in "abs.gram" describes the structure of itself. The use of the same editor for both programs and grammars gives a consistent user interface throughout the environment.

The grammar interpretive technique allows the user of the environment to experiment with languages in an explorative way. This may be used for several purposes. One is implementing an existing language, e.g. Simula or Pascal. The validity checking of the grammars is supported both by the structure-oriented editing and by obtaining fast feedback on changes. The grammars may in this way be "debugged" interactively similarly as code is debugged when editing and executing a program. The environment also makes it easy to adapt a grammar to the evolution of a language or to a specific dialect. Another area is language prototyping. When developing new programming languages or formalisms the advantages of rapid prototyping are obtained. Suggestions for language constructs may be described in the grammars and the result may be evaluated interactively. This may, for instance, be used for designing formalisms, new programming languages, and small specialized languages for various applications, so called "application languages" or "little languages" [Bentley 86].

5. Design Issues

In this section we summarize a number of important decisions we have made during the design of the ORM environment. These are issues that have to be addressed in the design of any development environment. Some of the aspects have been covered earlier in this paper while most issues are discussed in depth in other publications [Gustavsson 89 & 90, Hedin 86,88 & 89, Minör 87, 88 & 90, Oscarsson 87 & 89, Taube 87]

Structure Based - The decision to represent and store programs as structures rather than text has been beneficial. It has been essential in keeping track of changes and realizing the fine grained incremental compilation we were heading for. Functionality can directly take advantage of this representation and work according to the language definition. We have settled for using structure editing because it enables the system to give support in all situations. There are situations where keyboard typing still seems more efficient to use (entering expressions is one) which is always pointed out by experienced programmers. Although we currently provide also conventional parsing of expressions as an alternative, we still find it stimulating to try to find ways to improve the interaction.

Incremental Compilation - Our outset to support development of large industrial systems has ruled out interpretation for program execution because of efficiency and demand for early error detection. The challenge has been to make the semantic analysis responsive enough not to disturb the user. It has been essential to design the data structures to reduce the part of the program needing re-checking after a change. The result is that the scope of a change can be limited to where it actually has an impact. This is done with little overhead in searching or in stored information.

Partitioning - In order to support development of large applications we have put effort into finding solutions that can be believed to scale up nicely with program size. One of the key ideas here has been to focus on the *block* structure present in many programming languages in the form of procedures, classes, modules etc. These are important for the compilation since they often define new name scopes and from the user point of view as representing an abstraction. The internal data structures reflects this observation as well as structures on persistent storage. Information is stored on a block basis with the motivation that this matches the access patterns.

Integration - This has been a key concept and has been applied at different levels. The user interface presents the ORM environment as a single tool. All functionality is activated by need (semantic checking, code generation,etc.) without user interference. We also have the view that all functionality needed in the program development process should be integrated: documentation, version control, etc. It should all benefit from the structural approach. The system is open ended to support integration of new functionality. We have also developed a special approach for the communication between functional components.

User Interface - In order to make all the functionality manageable for the user we have settled for a graphical, direct manipulation user interface. An approach that has been successful for other applications. The user interface emphasizes the notion of blocks by showing every block as a separate window. Nesting in the programming language is shown as nesting of windows, a notion that has been specially developed in the Mjølner project. Blocks are the major abstraction mechanism used in programming languages. The effect of having each block appear as a separate window is that the user will directly manipulate these abstractions.

Language dependence - In ORM we have taken the approach of having components that interpret specifications of the language dependent parts. The interpretive technique gives a number of advantages; quick "turn around" when changing the specifications, tailorability, and extensibility. The possible disadvantage of too slow response time has not been a problem. The fact that we use a fine-grained incremental compilation scheme, especially for static semantic checking, is probably an important reason for this positive result.

Incremental execution - One of the long range goals in the project is to support development of real-time applications. In such situations the possibility to change programs on the fly is essential. The ORM environment has been designed to make this possible although the special problems of interaction with a running program in real time has not yet been dealt with. The real time aspect has influenced the designed on several points: The strive for efficient execution of the developed programs has been more pronounced. The run time structures have been designed to allow incremental loading and exchange of the generated code. The developed program and the environment run in a host-target configuration, possibly on different computers. Finally we have designed a garbage collection algorithm with guaranteed response time. We believe that these aspects are also valuable when developing traditional programs in an exploratory fashion although one could do with less demanding solutions.

Version/variant/configuration control - No real program exist in only one version. We have therefore chosen to include functionality for version, variant and configuration control already from the beginning. We also find these aspects essential when supporting teams of programmers working together. For program storage, a specialized storage scheme supporting tree structures has been designed. The use of general relational databases initially sounds like an attractive solution but has been reported [Linton 84] to give unacceptable performance. In our database information about one block is stored together. Most bindings are local to a block and bindings between blocks are especially encoded. This two-level scheme reduces the number of references that has to be supported by the data base. This leads to reduced complexity which should be valuable when handling large programs.

The development of ORM has served several purposes. One is to actually implement and try out the techniques mentioned above. Another was to establish a platform from which we can go on to integrate more advanced features needed for programming in the large and covering more of the software life cycle. An alternative would have been to stay compatible with the traditional textual representation. This is currently the dominating trend for commercial systems which are often built on existing tools as a way to quickly get useful results. We are, however, convinced that in the long run incremental techniques based on a structured representation have a much higher potential for achieving powerful program development systems.

6. Implementation

A cornerstone in the implementation of ORM has been the focus on block structure. A block is a generalization of the abstraction constructs in programming languages, e.g. procedures and classes. Most programming languages (except perhaps for logic-oriented languages like Prolog) exhibit a block structure, either unlimited as in Algol and Simula, or with certain limitations such as in Smalltalk (methods are nested within classes, but neither classes nor methods can be nested inside a method), and C (functions cannot be nested inside other functions).

In our opinion the block structure is of distinguished importance both from the user's point of view and from the implementation point of view. This is reflected in ORM in the user interface, the source program representation, the runtime representation, and the program database.

User interface

In the user interface the presentation is focused on blocks and block instances, using hierarchical windows to visualize the context in terms of block nesting as described in section 3. This interface has been implemented in HOPE, an object oriented programming interface to windowing systems [Oscarsson 87, Hauge et al. 88, Oscarsson 89]. HOPE is general purpose and written in Simula, currently using NeWS as the bottom layer. Its key features are hierarchical overlapping windows, light-weight processes (built on coroutines of Simula) and event-driven communication. The use of processes contributes to the modelessness of the system: Interaction can continue even if the system is busy with some computations and "mini-modes", such as filling in promptboxes, lock only the responsible window.

Source program representation

A source program is represented as a *block tree*, where each block object contains local information such as symboltable, abstract syntax trees for locals and body, local code generation information, etc. Each block is manipulated by a number of cooperating component instances: structure-oriented editor, presentation component, semantic checker, and code generator. In order to make the environment open-ended the components communicate using a notifying scheme [Minör 88] which enables various different configurations without affecting the components themselves.

Structure-oriented editor

Both the syntax trees of a block and the syntax trees representing aspects of a grammar are edited using the structure-oriented editor. The editor is based on structure synthesis and direct manipulation interaction rather than parsing structures from text. The editor manipulates a syntax tree according to an abstract grammar. This grammar is consulted during each edit operation in order to guarantee syntactical consistency. This is described in more detail in [Minör 90].

Presentation component

The presentation component is responsible for presenting the syntax-trees. Whenever a syntax tree is modified the presentation component is notified about the change in order to incrementally update the presentation accordingly. This is done by interpreting a concrete grammar which specifies the presentation and layout of each construct in the syntax-tree. Furthermore the presentation component is responsible for presenting attributes, e.g. error attributes from the semantic checker, and for maintaining a mapping between screen locations and syntax-tree nodes. Concrete grammars and presentation is described in [Minör 90].

Semantic checker

The semantic checker incrementally maintains syntax-tree attributes and symboltables after each modification performed by the editor component. Incremental attribute evaluation techniques are used which consider attribute dependencies within a local abstract syntax tree. Non-local dependencies, e.g. dependencies between the syntax tree attributes and the symboltable, and between different symboltables in the block tree, are handled by general scope mechanisms which are interfaced to the attribute evaluation mechanism [Hedin 88]. This has been found a good solution as an alternative to the pure attribute grammar approach which does not work well for incremental scope handling. We have found other solutions to these problems, e.g. [Hoover et al. 86] to be insufficient for the combined inheritance and nesting scopes found in object-oriented languages. The semantic checker interprets evaluation plans which are generated from the semantic grammar.

Code generator

Code generation is done incrementally with the granularity of a block. Each block results in a template (a structural description of the block) and a code object (a sequence of instructions). Since there is no need for feedback to the user it can be run as a low-priority background process keeping the executable code consistent with the source representation. The code generation is divided into a front-end that translates syntax trees annotated with semantic attributes to an intermediate code and a back-end that translates the intermediate code to native code. The front-end interprets attribute evaluation plans generated from the code generation grammar. The back-end is created using a traditional generative approach. It is consequently possible to modify the target language interactively whereas generating code for a new target machine requires re-generation of the back-end.

Runtime representation

The runtime representation of a program is a *template tree* where each template corresponds to a block in the source representation. The dynamic state of the running program is represented as a collection of block instances which are all heap allocated and subject to garbage collection. Block calls (e.g. procedure calls, and class "new"s) are performed via indirect pointers in the template tree. The code for a block call is thus independent of the position of the called block. The flexibility of the template tree runtime representation is of great importance in the incremental setting. Individual templates and code objects can be incrementally loaded without affecting existing code in the system [Magnusson 83]. This is utilized to support incremental execution, i.e. allowing continued execution after changes to the program, mentioned in section 4.

The executing program runs as a UNIX process separate from ORM, enabling program development to take place in a host-target setting. This is an essential requirement for real-time systems which usually run on dedicated machines. Runtime observation (i.e. viewing the objects and procedure activations as described in section 4) and incremental loading is served by a small kernel embedded in the runtime system. There is thus no need for a separate production environment.

Since ORM is planned to handle programs with strict real-time requirements, the garbage collector has to work incrementally and guarantee free memory to be supplied at a required rate. Such demands are difficult to meet at a low cost. At the expense of a somewhat longer guaranteed response time, the garbage collection efficiency can, however, be improved considerably by using a generation-based algorithm [Bengtsson 90].

Program database

A program is stored in a program database on a block basis. Individual components of a block, such as the symboltable, or the abstract syntax tree for the locals, can be retrieved from the database in one operation. This grain of storage fits the access patterns of the user and of the compilation components. The database [Gustavsson 90] is tree structured and allows all kinds of data to be stored, e.g. syntax trees, symbol tables, window status, documentation, and generated code. Furthermore, programs, grammars, and other objects in the database (so called software objects) are subject to version control and configuration management [Gustavsson 89].

Implementation status

ORM is implemented in Simula and runs on SUN workstations. As a programming environment, the system is currently attractive mainly for motivated users. Programs up to the size of 1000 lines have been implemented. As a language laboratory, the system has proven highly attractive also for moderately motivated users, although it still has some rough edges. It has been used in teaching compiler construction courses. The largest grammar implemented is the one for Simula, consisting of about 100 productions. In handling full programming languages there are currently some shortcomings, mainly due to some semantic functions not being general enough. For this reason it has not yet been possible to implement e.g. full BETA in the system. Currently we are working on removal of rough edges, with the immediate goal of using the system in teaching introductory programming. Long term future plans include working towards realtime and programming in the many and in the large as well as defining grammars for other typed object-oriented languages, such as complete BETA [Kristensen et al. 87], Eiffel [Meyer 87], and C++ [Stroustrup 86].

The response times of the environment as a whole are typically fractions of a second. When executed on a modern workstation, most operations, such as edit operations, semantic checking, and changes to the language grammars, are fast enough for interactive use. The response times have not degraded noticeable when programs and grammars grow up to the sizes we at present have used as described above. As a matter of fact, the underlying window system at present is one of the most time-consuming components. We are thus confident that the approaches and techniques used scale up to support development of large programs in industrial applications.

7. Conclusion

We have presented ORM, an environment for program and language development. A significant characteristic of ORM is its consistent user interface based on direct manipulation. Direct manipulation relieves the user of cognitive burdens and has proven highly successful within the office automation area. We think it is extremely useful also for simplifying the complex activity of software development.

To make mundane activities such as compiling, linking, and loading fast, automatic, and invisible to the user, ORM employs incremental techniques which are triggered by need. Similar ideas have been pursued also in other environments such as Gandalf and the Cornell Synthesizer, but have been driven even further in ORM. ORM also differs from these environment in focusing on object-oriented languages which require more complex compilation technology, and by providing more powerful support during execution.

As a language development environment, ORM offers similar advantages as for programming: grammars are developed by means of direct manipulation and new or changed language constructs can be immediately tried out on a target program. This applies not only to developing programming languages, but to developing any formal language, including the metaformalisms themselves. Development of formalisms will certainly be an increasing area of importance within software development, in particular for the design and development of small special purpose languages.

For many important issues, like programming in the many and programming in the large, we have only scratched the surface. However, we believe it much more beneficial to tackle these problems within a highly interactive and structure-based environment like ORM, than within the traditional text-based environments.

References

[Bengtsson 90] Bengtsson, M., *Real-Time Garbage Collection*, forthcoming thesis, Dept. of Computer Science, Lund University, 1990

[Bentley 86] Bentley, J., *Little Languages*, Communications of the ACM, Vol. 29, No. 8, 1986

[Borras et al. 88] Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V., *CENTAUR: the system*, Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments, November, 1988

[Conklin 87] Conklin, J., *Hypertext: An Introduction and Survey*, IEEE COMPUTER, September, 1987

[Dahle et al. 87] Dahle, H.P., Löfgren, M., Madsen, O.L., Magnusson, B., *The MJØLNER Project*, Proceeding of the Conference held at Software Tools, Online Publications, London, 1987

[Dart et al. 87] Dart, S.A., Ellison, R.J., Feiler, P.H., Habermann, A.N., *Software Development Environments*, IEEE COMPUTER, November, 1987

[Delisle et al. 84] Delisle, N.M., Menicosy, D.E., Schwartz, M.D., *Viewing a Programming Environment as a Single Tool*, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 1984

[Donzeau-Gouge et al. 80] Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., *Programming Environments Based on Structure Editors: The MENTOR Experience*, INRIA Research Report, No. 26, 1980, also in Barstow et. al. (eds.), *Interactive Programming Environments*, McGraw-Hill, 1984

[Feiler et al. 86] Feiler, P.H., Jalili, F., Schlichter, J.H., *An Interactive Prototyping Environment for Language Design*, Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, 1986

[Goldberg 84] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley Publishing Company, 1984

[Gustavsson 90] Gustavsson, A., *Software Configuration Management in an Integrated Environment*, Thesis LU-CS-TR:90-52, Dept. of Computer Science, Lund University, 1990

[Gustavsson 89] Gustavsson, A., *Viewing the Evolution of Software Objects in an Integrated Environment*, Proceedings of the Second ACM SIGSOFT Workshop on Software Configuration Management, November, 1989

[Hauge et al. 88] Hauge, T., Nordgard, I., Oscarsson, D., Ræder, G., *Event-Driven User Interfaces Based on Quasi-Parallelism*, 1st ACM Symposium on User Interface Software, 1988

[Hedin et al. 86] Hedin,G., Magnusson, B., *Incremental Execution in a Programmning Environment Based on Compilation*, Proceedings of the 19th Hawaii International Conference on System Sciences, 1986

[Hedin et al. 88] Hedin, G., Magnusson, B., *The Mjølner Environment: Direct Interaction with Abstractions*, Proceedings of ECOOP '88 European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, Vol. 322, Springer-Verlag, 1988

[Hedin 88] Hedin, G., *Incremental Attribute Evaluation with Side-effects*, Proceedings of the Workshop on Compiler Compiler and High Speed Compilation, Berlin GDR, 1988

[Hedin 89] Hedin, G., *An Object-Oriented Notation for Attribute Grammars*, Proceedings of ECOOP '89 European Conference on Object-Oriented Programming, 1989

[Henderson et al. 87] Henderson, P.B., Notkin, D., *Integrated Design and Programming Environments*, IEEE COMPUTER, November, 1987

[Hoover et al. 86] Hoover, R., Teitelbaum, T., *Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars*, Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, 1986

[Hutchins et al. 86] Hutchins, E.L., Hollan, J.D., Norman, D.A., *Direct Manipulation Interfaces*, in Norman, D.A., Draper, S.W. (ed.), *User Centered System Design*, pp. 87-124, Lawrence Erlbaum Associates Inc., 1986

[Johnson et al. 89] Johnson, J., Roberts, T.L., Verplank, W., Smith, D.C., Irby, C.H., Beard, M., Mackey, K., *The Xerox Star: A Retrospective*, IEEE Computer, September, 1989

[Knudsen et al. 89] Knudsen, J.L., Madsen, O.L., Nørgaard, C., Petersen, L.B., Sandvad, E., *An Overview of the Mjølner BETA System*, Mjølner Informatics ApS, Aarhus, 1989

[Kristensen et al. 87] Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B., Nygaard, K., *The BETA Programming Language*, appering in Shriver, B., Wegner, P., *Research Directions in Object-Oriented Programming*, The MIT Press, 1987

[Linton 84] Linton, M.A., *Implementing Relational Views of Programs*, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, Vol. 19, No. 5, 1984

[Magnusson 83] Magnusson, B., *Code-Objects: a Support for Incremental Compilation*, Department of Computer Science, Lund University, 1983

[Meyer 87] Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, 1987

[Minör 87a] Minör, S., *A Generic Synthesizer and its Implementation*, Proceeding of the Fifteenth SIMULA Conference, Jersey, 1987

[Minör 87b] Minör, S., *Structured Command Interaction Based on a Grammar Interpreting Synthesizer*, Proceedings of the Second IFIP Conference on Human-Computer Interaction, North-Holland, 1987

[Minör 88] Minör, S., *Flexible Communication Between Objects*, Proceedings of the Sixteenth SIMULA Conference, Innsbruck, 1988

[Minör 90] Minör, S., *On Structure-Oriented Editing*, PhD thesis, Dept. of Computer Science, Lund University, 1990

[Møller-Pedersen et al. 87] Møller-Pedersen, B., Belsnes, D., Dahle, H.P., *Rationale and Tutorial on OSDL: An Object-Oriented Extension of SDL*, Computer Networks, Vol. 13, No. 2, North-Holland, 1987

[Notkin 85] Notkin, D., *The GANDALF Project,* The Journal of Systems and Software, Vol. 5, No. 2, May, 1985

[Oscarsson 87] Oscarsson, D., *A Quasi-parallel Process Concept to Ease the Use of Event Driven Interfaces from a Program*, Proceedings of the Fifteenth Simula Conference, Jersey, 1987

[Oscarsson 89] Oscarsson, D., *HOPE : Hierarchical Object-oriented Presentation Environment - Reference Manual*, Dept. of Computer Science, Lund University, 1989

[Reiss 85] Reiss, S.P., *PECAN: Program Development Systems that Support Multiple Views*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 3, March 1985

[Reiss 87] Reiss, S.P., *Working in the Garden Environment for Conceptual Programming*, IEEE Software, November, 1987

[Reps et al. 84] Reps, T., Teitelbaum, T., *The Synthesizer Generator*, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Environments, 1984

[Sandewall 78] Sandewall, E., *Programming in an Interactive Environment: the "LISP" Experience*, Computing Surveys, Vol. 10, No. 1, March, 1978

[SIS 87] SIS Swedish Standard , *Programming languages - SIMULA*, SIS, Stockholm, 1987

[Stroustrup 86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986

[Taube et al. 87] Taube, M., Hedin, G., Magnusson, B., *The MJØLNER Observation Tool: An Object Oriented View of Programs and Program Execution*, Proceeding of the Fifteenth SIMULA Conference, Jersey, 1987

[Teitelman et al. 81] Teitelman, W., Masinter, L., *The Interlisp Programming Environment,* IEEE Computer, Vol. 14, No. 4, April, 1981, also in Barstow et. al. (eds.), *Interactive Programming Environments*, McGraw-Hill, 1984

[Winograd 87] Winograd, T. A., *Muir: A Tool for Language Design*, Report No. STAN-CS-87-1151, Department of Computer Science, Stanford University, 1987