Using Mjølner Orm as a
Structure-Based Meta Environment

Sten Minör and Boris Magnusson

LU-CS-TR:92-101

To be published in L. Neal and G. Szwillus (eds)
Structure-Oriented Editors and Environments, Academic Press, 1994, to appear

Department of Computer Science
Lund University

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

Using Mjølner Orm as a Structure-Based Meta Environment

Sten Minör and Boris Magnusson
Department of Computer Science
Lund University
Lund, Sweden

Abstract

Mjølner Orm is a structure-based environment supporting incremental software and language development. The environment functionality is specified by grammars developed to support object-oriented languages. These grammars are interpreted by the environment allowing both the syntax and semantics to be changed on the fly. This has been shown useful for language customization, design, and maintenance. This paper gives an overview of the environment and focuses on Orm used as a meta environment.

1 Introduction

The problems with developing software systems are widely acknowledged. After several decades of software development experience there is still no satisfactory solution for the "software crisis". There are no "simple" solutions or shortcuts to these problems and it has to be accepted that development of large software systems is a complex and time-consuming task. We do, however, believe that providing appropriate support for software engineers will make the software development process far more efficient than it is today.

The overall goal in the Mjølner project is to provide support for object-oriented software development in general. The project started in 1986 as a joint Nordic project involving universities and industry in Denmark, Finland, Norway, and Sweden. The activities within the project have dealt with three different aspects of software development:

- *Methods* for object-oriented software development
- Object-oriented *languages*
- *Environments* for object-oriented development

The method activity has worked with support for specification. It has resulted in the design of an object-oriented extension to SDL, called OSDL [Møller-Pedersen et al. 87]. OSDL has after smaller modifications been accepted by CCITT as the new standard, SDL 92.

The language aspect has been covered in the development of the language Beta [Kristensen et al. 87]. Beta, a simple but powerful object-oriented language, is a successor of Simula [SIS 87], generalizing its constructs for abstraction and inheritance.

The environment activity includes the development of the Mjølner Orm environment and the Mjølner Beta environment. The Beta environment [Knudsen et al. 89] has been developed along with the work with the Beta language at Aarhus university in Denmark. The focus of this paper is the Orm environment, developed at Lund University in Sweden. Orm is a generic structure-oriented environment supporting object-oriented languages.

The work with the Orm environment has covered a number of topics within the area of software development environment research, e.g. structure-oriented architectures, structure editing, incremental compilation techniques, configuration control, user interfaces, grammar formalisms, and grammar interpretation. In the next section, section 2, we will give an overview of the environment. After an introduction, the system is presented from the user's point of view in section 2.1. Section 2.2 shows the user's view of the environment used for program development. The functionality of the environment and the architecture is presented in section 2.3. Orm as a meta environment is discussed in section 3. Section 3.1 shows the user's view of grammars and grammar editing. Section 3.2 describes the grammar interpretive technique which is the basis for the meta capabilities. The practical consequences of the grammar interpretive technique is discussed in section 3.3. In section 4 the implementation status and experience from using Orm in practice is presented, followed by some topics for future research in section 5 and a summary in section 6.

Mjølner (pronounced something like "Miolner") is a name from the ancient Nordic mythology. It is the name of the hammer of the god Thor. Mjølner is his most powerful tool. It always hits its target, and afterwards it returns to Thor's hand. When Thor uses his hammer, it is noticed by humans as thunder and lightning. Orm is the name of the main character in a famous Swedish saga of the viking age. He was a great admirer of Thor, even after he eventually, against his will, was christened.

2 The Orm Environment

Common interests in the Mjølner project has been a strive to support industrial large scale software development and object-oriented programming. The participants in the project have a long tradition of using object-oriented languages, even before the term was invented and became popular. In the Scandinavian tradition, block-structured, strongly typed object-oriented languages play an essential role [Madsen et al. 90]. Supporting early error detection, such as static typing, is important when developing high-quality software. Turn around time in systems for such, usually compiled, languages has, however, often been a major problem. This is in contrast to interactive environments based on interpretation which often show 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.

Another problem in traditional, so called "tool based", environments is the proliferation of tools, all with different user interfaces. In the Orm environment we have designed a direct manipulation graphical user interface [Hutchins et al. 86] used consistently in program manipulation, observation of program execution, and for the definition of language grammars. In the Orm environment the functional components are integrated into one single tool. The components use a shared program representation, which is built on program structures rather than source text. This makes it possible to add functionality to the system, sharing the structured view of programs. Extensibility 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 structural view is also crucial for the development of fine-grained incremental compilation techniques and more suitable as basis for language-specific support such as structured browsing, navigation tools, as well as syntax-based and semantics-based editing than a textual representation.

The Orm environment is generic and can be specialized to support different languages. The technique used for specialization is interpretation of grammars defining the actual programming languages. The grammars have been developed with Orm itself which shows to us the value of interactive development in language implementation as well. A modification of 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, supporting special syntax, or even using Orm as a language laboratory.

Several structure-oriented environments have address similar problems before. Mentor [Donzeau-Gouge et al. 80] is an early example. Other structure-oriented environments include The Synthesizer Generator [Reps et al. 84], Gandalf [Notkin 85], PSG [Bahlke et al. 92], DOSE [Feiler et al. 86], Muir [Winograd 87], Centaur [Borras et al. 88], and Pan [VanDeVanter et al. 92].

The major contributions of the Orm environment are:

- Consistent use of a hierarchical object-oriented browser to represent different tree-structures such as: nested program constructs, runtime data structures, file system directories, and grammar aspects
- Direct manipulation of objects in an integrated environment rather than tool interaction
- Incremental compilation
- Graphical object-oriented execution observation
- Interactive development of language grammars with immediate feedback
- 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

2.1 A structured view of the system

In order to give an overview of the system, we start with the user's perspective. 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.

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 in a particular location, which is usually significant for its meaning. For instance, the meaning of a procedure (method) "draw" depends on which class it resides in. For some kinds of objects, the location is more of organizational than of semantic importance such as the organization of files into different folders. This physical structure is visualized directly in the Orm browser by means of a generalized window system. In traditional window systems the windows reside on the screen, overlap and can be iconized to save screen space. In the Orm window system this principle is generalized to allow not only the screen to 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. For instance, a class containing 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.

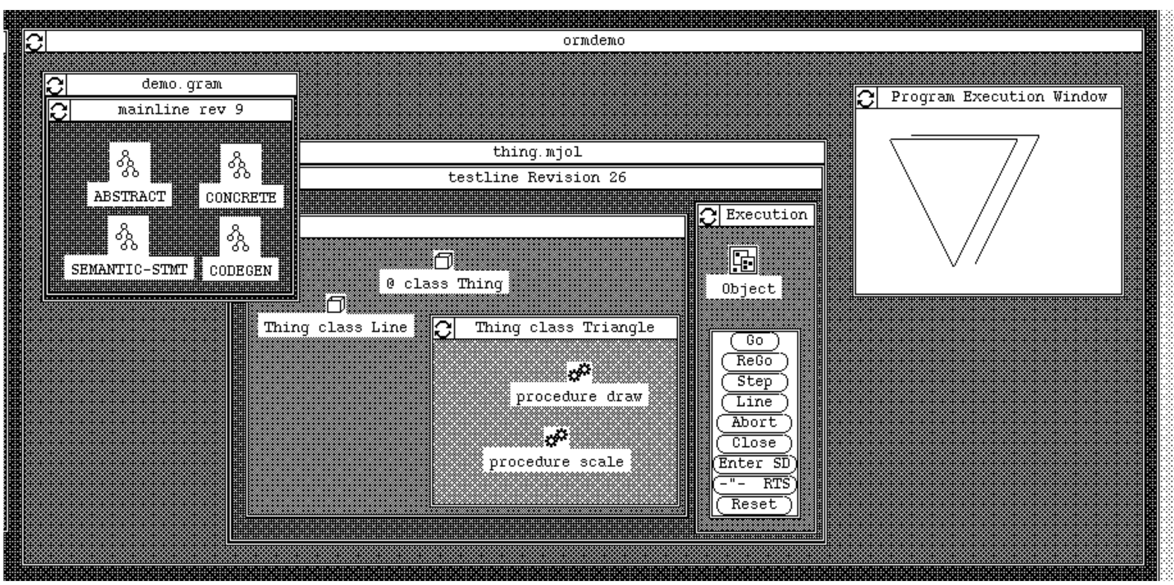

FIGURE 1. The user's view of the Orm system at the top level

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 browser 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 or moved. When iconizing the program window, all its parts (main block, local classes and procedures) will disappear from view, but reappear in exactly the same state when the program window is opened again. Such information is preserved also between sessions.

In addition to the structural navigation supported by the browser, there are many important relations along which the user may want to navigate. 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 to bring another part of the hierarchical structure into view quickly.

The direct manipulation interface of Orm is similar in flavor to the Macintosh finder. It differs from the Macintosh finder by the introduction of hierarchical windows which always shows an object (window) in its context. We consider this a particularly valuable property in programming environments for object-oriented languages where the context has strong semantic importance. Earlier programming environments utilizing graphics and windows usually follow the tool-based approach where the windows represent tools rather than objects, like for instance the browser and inspector tools in the Smalltalk environment [Goldberg 84].

2.2 A structured view of a program

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 executing a program. 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 two 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 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 point of the error.

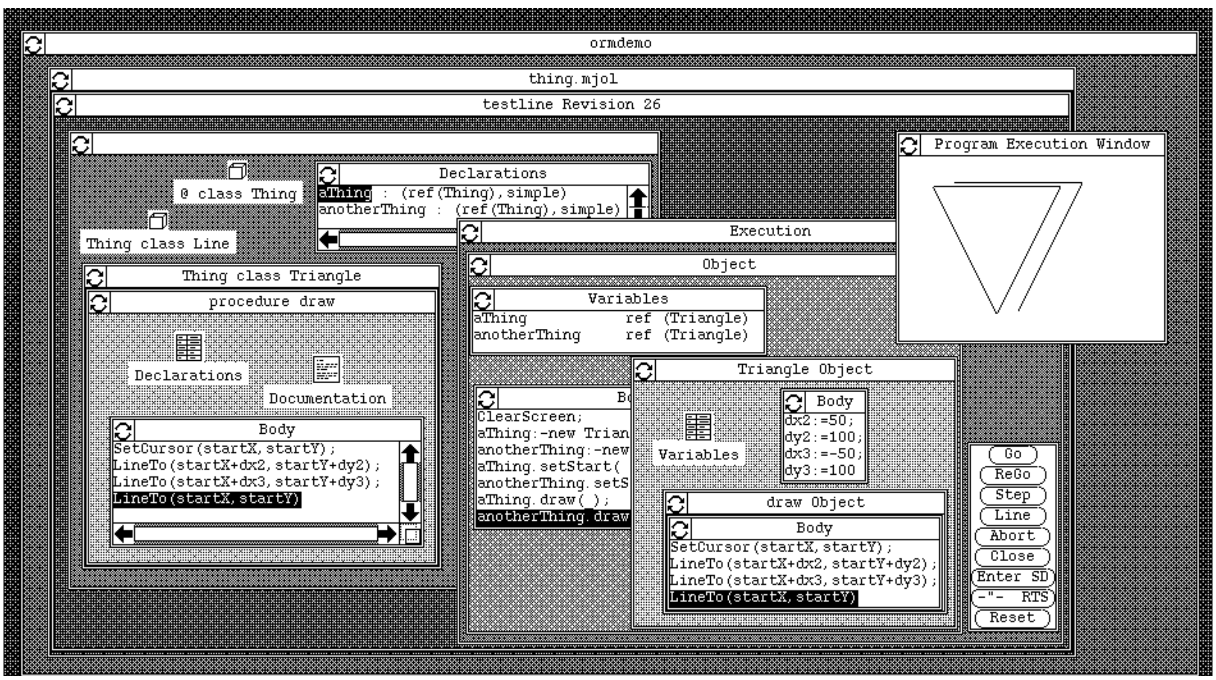

FIGURE 2. A program in Orm

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 within the object. 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 the types as is the case 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 it works like a link button which links to the referenced object. When the user clicks on the variable “anotherThing” for instance, the “Triangle” object's window flashes. For simple variables, such as integers and booleans, the value is shown directly in the local variables window.

The link facility for reference variables provides a method of navigating within object structures. In our example, the variable “aThing” references a Triangle object other than the one currently shown. This object will appear when the user clicks on the variable. Another way to bring up windows is via the call stack (not shown in the example). The user thus has full control over what objects are shown in the execution window. He can iconize and also remove windows as he wishes. The reference variables act as starting points for exploration by the user.

2.3 Functionality in Orm

Orm is a structure-based environment, representing programs as well as grammars as attributed abstract syntax trees. The representation is shared among a number of tightly integrated components. The components together form the functionality of the environment. The components are integrated in two ways. First, they are communicating using a scheme for notifications, which allow incremental updates of changes to the shared representation to be distributed to other components. The notifications minimizes a component’s knowledge of other components in the environment resulting in an open-ended environment where components can be added and removed without changes to the existing components [Minör 90]. Second, the components share the same user interface, appearing as one single, seamless tool to the user.

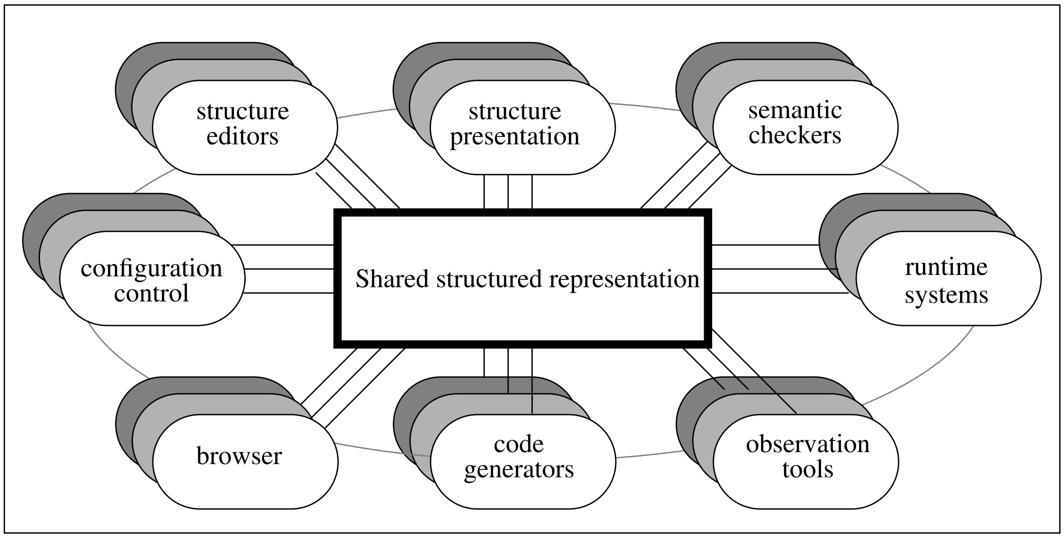

FIGURE 3. The components of the Orm environment

Figure 3 shows a simplified view of the components in the environment. Both grammars and programs are represented as abstract syntax trees. The abstract syntax trees are divided into two levels. The higher level represents *blocks*; a block is typically an abstraction in a programming language, e.g. a class, a method or a procedure, or an aspect of a grammar. The lower level represents the contents of a block, e.g. a class body, the contents of a method or procedure, or the contents of a grammar aspect.

The division of the abstract syntax tree has been used throughout the system. The block level is the basic means for navigation in the system using the hierarchical browser. It is also used by the incremental compiler for maintaining symbol tables and defines the granularity of the code generation. A program is organized as blocks also when stored in persistent storage in a program database. 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 single 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. Finally, the block level has been the fundament for the architecture of the environment. Each block has bound to it a cluster of communicating components operating on its content, e.g a structure editor, a presentation component, a static semantic checker, and a code generator. The clusters of components bound to different blocks are executing in parallel allowing multiple simultaneous activities in the different blocks, such as simultaneous editing of different classes and procedures.

The remainder of this section will give a short overview of the research areas involved in the Orm environment.

Structure editing

Since Orm is a structure-oriented environment, both programs and grammars are edited by means of structure editing. There are two levels of structure editing corresponding to the division of the representation into the block level and the block-contents level.

At the block level, a structure is viewed as a hierarchy of windows in the browser [Hedin et al. 88]. The browser is essentially a structure editor with a semi-graphical presentation. The browser is used for editing the structure in accordance with a block grammar. The block grammar is defining the top level of a language's syntax. A program, for instance, consists of a number of classes and procedures and one body. A class, in turn, consists of instance variables, a class body and a number of procedures. The browser allows the user to navigate in a program in terms of abstractions of the target language. The hierarchy of the abstractions are made explicit by the window hierarchy. In contrast to browsers in most other environments, e.g. the Smalltalk browser which is activity-oriented, the Orm browser is object-oriented, allowing an arbitrary number of blocks to be open and edited simultaneously.

Within a block, a nested structure editor is used for editing. The structure editor uses textual presentation of the block's contents. The contents is edited in accordance with the syntax of the language which is expressed in language grammars. The main contributions of the Orm structure editor, compared to other structure editors, is the consistent representation of grammars and programs as abstract syntax trees and the grammar-interpreting technique utilized for defining language dependence. This is described more in detail in [Minör 90] and we will expand on it somewhat in section 3. The editor is basically a "pure" structure using the "synthesizer approach", although we currently provide also conventional parsing of expressions as an alternative. However, we work on finding alternative ways to improve the interaction in pure structure editors and avoid parsing of text altogether [Minör 92].

The editor supports *context-sensitive editing* [Hedin 91]. Based on the semantic information available in the semantic attributes in the abstract syntax tree, e.g. symbol tables, scope rules, and types, the editor presents variables, classes, and procedures visible in the current scope in a menu. Typed references in statically typed object-oriented languages makes it possible to determine the legal set of messages to an object. The messages are made available in submenus and will, if selected, be inserted in the program with the correct number of parameters. This facility has proved to be very practical and is appreciated by the users. Particularly when using class libraries or tool-boxes, context-sensitive editing relieves the user from remembering the class names, their methods, and the protocols for calling them.

Incremental compilation

Orm is based on incremental compilation, for two reasons. One reason is efficiency. Since the techniques used in Orm are intended to be useful in "real" applications, the efficiency of the executing program is important. Thus compilation to native code is required. The turn-around time from editing to execution must not be too long in an interactive system.

Thus fine-grained incremental techniques are used. The second reason for incremental compilation is early error detection. If the user makes an error, such as a creating a type system inconsistency, the environment should report this error as soon as possible, preferably immediately after the editing operation causing the error. The user can then either correct the error or leave it as it is for later correction. The incremental compilation is divided into two components: the incremental static semantic checker and the code generator.

A semantic checker incrementally maintains syntax-tree attributes and symboltables after each modification performed by an 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. 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 a semantic grammar. More details on the semantic checking in Orm and the attribute grammar formalism are given in [Hedin 92].

Code generation is done incrementally with the granularity of a block, i.e. a class, procedure or equivalent. 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 medium-grained incremental code generation lies in between the traditional separate compilation of modules and the fine-grained incremental compilation technique presented in [Fritzson 84]. The code generation is divided into a front-end translating syntax trees annotated with semantic attributes to an intermediate code and a back-end translating 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. Hence, it is possible to modify the target language interactively whereas generating code for a new target machine requires re-generation of the back-end.

Run-time environment

The runtime environment is not based on formal descriptions. It is hand-crafted using more traditional techniques. However, it has been our purpose to make the design general enough for runtime representation of programs in a class of block-structured object-oriented languages, such as Simula [SIS 87], Beta [Kristensen et al. 87], Eiffel [Meyer 87], and C++ [Stroustrup 86].

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 [Hedin et al. 86].

The executing program runs as a UNIX process separate from the development system, 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 2.2) and incremental loading is done by a small kernel embedded in the runtime system. The bulk of the code implementing this functionality is part of the development system.

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].

Configuration control

The configuration management component of Orm basically has two purposes. It supports the user in controlling the evolution of (large) software systems, and it defines and implements a data model used for persistent storage of programs and grammar structures.

The purpose of the data model is to store the block tree and the contents of the blocks, i.e. syntax trees and block attributes, on persistent storage. The block attributes may contain information such as symbol tables and generated intermediate code created by various components in the environment. Attributes of the abstract syntax tree nodes may refer to the block attributes, and the block attributes may refer to other blocks in the block tree. The data model has to deal with is a graph structure, even though the basic structure is tree-like. The physical storage of the structure is implemented as sequences of bytes stored in Unix files. The representation on persistent storage is a byte representation of the program *structure* which has little in common with traditional the textual representation of programs. Furthermore, the configuration control component handles demand loading of program and grammar structures from persistent storage. In this way the space requirements for the abstract syntax tree representation is reduced since only the parts of the representation actually used are loaded.

The evolution control supports the user in keeping track of *versions* and *alternatives* of program and grammar structures. An alternative of a program is understood as a series of versions that represent one evolution line among other alternatives. The configuration manager supports the user in creating different alternative lines of evolution and their respective versions. It also supports several different binding rules between program modules and between programs and grammars. It guarantees mutual exclusion, i.e. that modifications are not done by two users simultaneously in an uncontrolled manner. It also offers ability to merge two alternative evolution lines into one. Different specializations of such objects make it possible to maintain program structures and to support organization of software objects by membership and by composition in a uniform way. This is described in detail in [Gustavsson 90].

3 Orm as a Meta Environment

Several components of the Orm environment are generic.Their language dependence are defined in grammars which are interpreted by the components. Both grammars and programs are represented and stored as abstract syntax trees in Orm enabling the use of the environment for language development as well as software development. When working on language grammars, the Orm components are interpreting meta grammars. In this section we explain how to use Orm as a meta environment. More details are found in [Minör 90].

3.1 A structured view of grammars

In section 2, the editing and execution of a program object within Orm was shown from the end user's perspective. Here, we will show the grammars from the language implementor's perspective.

Grammars are manipulated by "opening" a grammar connected to a program. Editing a grammar includes development of the syntax as well as the static and dynamic semantics of the language. Like a program, a grammar in Orm has 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 using different formalisms is a consequence of the goal to make Orm open-ended. This includes the possibility to add new language aspects if necessary and to make configurations of Orm where some aspects are not present. The architecture allows experiments with different formalisms for a certain language aspect without affecting the other aspects.

- The *abstract grammar* defines the abstract syntax.
- The *concrete grammar* defines the concrete syntax.
- The *semantic grammar* defines the static semantics
- The *code-generation grammar* defines the translation to intermediate code.

Figure 4 shows editing of the grammar "demo.gram". The four aspects constituting the internal structure of the grammar have all been opened and may thus be manipulated by means of structure-oriented editing. The fact that the grammar is interpreted means 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 program may be manipulated and executed according to the new version of the language, i.e. without leaving the environment or re-generate parts of the environment.

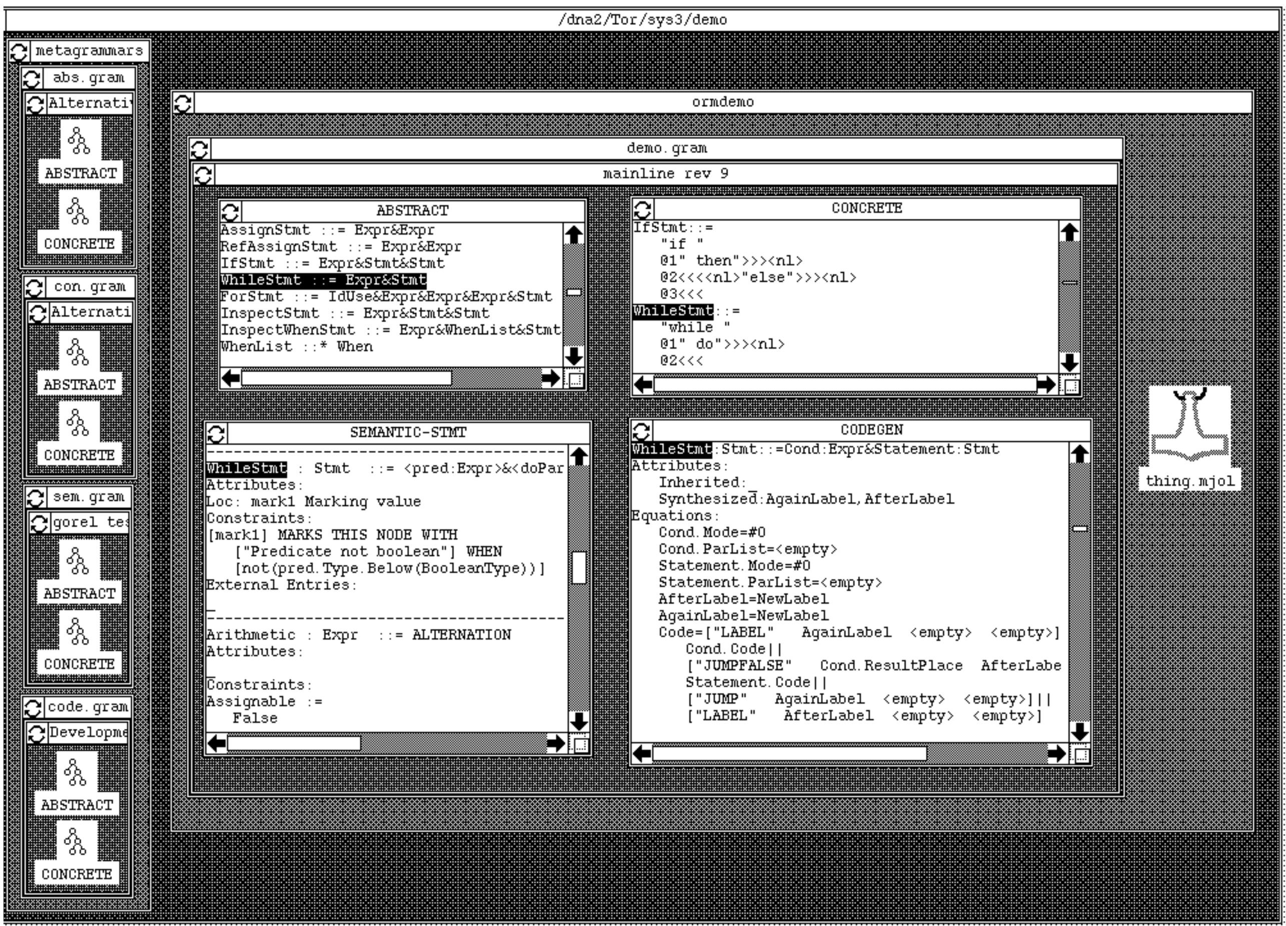

FIGURE 4. Editing a grammar

Grammars and programs are edited with the same structure editor. The editor interprets meta grammars when editing these language grammars. E.g. the meta grammar "abs.gram" in figure 4 describes different aspects of abstract grammars. Similarly, the other language grammars are described by a corresponding meta grammar. The use of the same editor and browser for both programs and grammars gives a consistent user interface throughout the environment.

3.2 Grammar interpretation

Earlier structure-oriented environments, such as the Cornell Synthesizer [Teitelbaum et al. 81], were "hand-crafted" for one specific language. They were followed by systems allowing structure-oriented editors or environments to be automatically generated from a language description. ALOE [Medina-Mora 82] and the Synthesizer Generator [Reps et al. 84] are two examples of such systems. In environments generated from language grammars the implementor alone is able to modify the grammars. Grammar modifications in generative systems are typically done in a “batch”-oriented manner with a considerable “turn-around” time from the grammar modification until a new generated environment is obtained.

Orm differs from most other structure-oriented environments by its grammar interpretive approach. Orm *is* a generic environment that is parameterized with language specific information by interpreting language grammars dynamically. A consequence of the 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. The ability to develop or modify grammars can either be utilized by a language implementor or by the end user.

Figure 5 illustrates the differences between a hand-crafted environment tailored for one language, environments generated from language grammars, and a generic grammar-interpreting environment

.

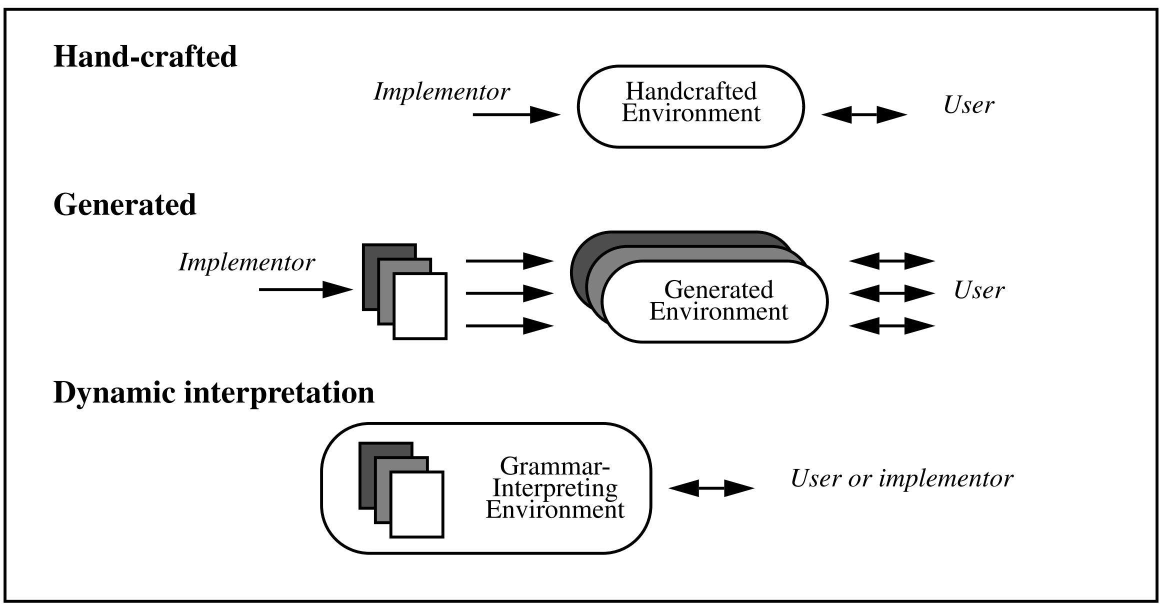

FIGURE 5. Different kinds of environments

Other environments based on interpretation of language descriptions are DOSE, developed at Carnegie-Mellon University within the Gandalf project, and Muir, developed at Stanford University.

DOSE [Feiler et al. 86], an acronym for Display-Oriented multi-purpose Structure Editor system, is a system mainly intended for language design. DOSE is interpreting grammars for syntax definitions and uses an action routine scheme for the semantics. The environment is divided into the implementor's environment which is generated from language grammars and the user's environment interpreting the language description.

Muir [Winograd 87] is a structure-oriented language development environment. The environment is interpreting syntax descriptions both at the target language level and at the meta level. The environment has functionality for transformation between abstract syntax trees derived from different versions of a grammar [Nørmark 87].

Orm, to our knowledge, is the only system with syntax, static semantics and code generation based on grammar interpretation. Programs and grammars are consistently represented as abstract syntax trees, and the same generic components are used, including the meta level.

Grammar interpretation in Orm

As mentioned in the previous section, a language is described by four grammar aspects. Each aspect is interpreted by different components in the environment. Figure 6 shows the components and the grammar aspects interpreted. In the block grammar it is defined how the grammar is divided into higher level blocks as discussed in section 2.3. Below, we will sometimes call the abstract aspect of a language grammar the "abstract grammar" for short, and likewise the concrete aspect of a grammar the "concrete grammar", etc.

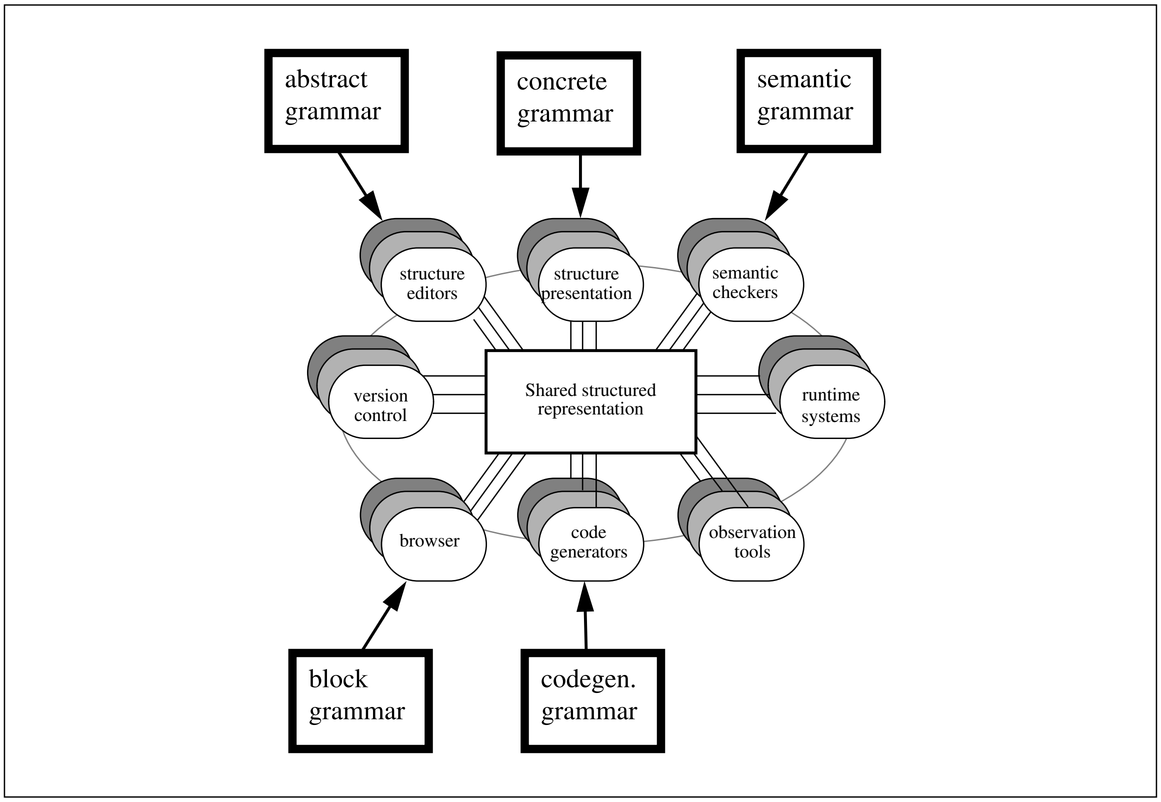

FIGURE 6. Grammar aspects interpreted by Orm components as yet

- The *block grammar* describes the structure of the higher levels of the abstract grammar, such as classes, procedures, etc.
- The *abstract grammar* defines the structure of the language. It is specified in a structured variant of extended BNF with all concrete syntax elements removed.
- The *concrete grammar* defines the textual presentation of the language constructs, as defined in the abstract grammar. This includes the specification of the “syntactic 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, called door attribute grammars [Hedin 92], allowing attributes and equations to be inherited between productions. This allows the behavior to be specified at the appropriate level of generalization. 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 expressing the dynamic semantics of the language. In this case an object-oriented variant of attribute grammars is used as well.

The browser interprets block grammars. It traverses the block grammar representation in order to check the editing operations at the block level. The block grammar is also the basis for constructing the language menu used for editing in the browser.

The abstract grammar is interpreted by the structure editor. The operations in the editor are: *expansion*, i.e. replacement of a meta node by a single new node, *pasting*, i.e. the replacement of a meta node by existing subtrees, *cutting*, i.e. the removal of subtrees, and syntactical *transformation*. In all those operations the abstract syntax tree representing the abstract grammar is traversed in order to check the consistency of the operation with the grammar. The menus used for structure editing are initially derived from the abstract grammar and the concrete grammar.

Interpretation of the concrete grammar is done by consulting the concrete grammar representation whenever the target structure has been modified by the editor. This is done by a presentation component, which is responsible for keeping the screen representation updated incrementally.

The semantic grammar is used by the incremental semantic checker components. A semantic checker maintains the attributes in the target tree incrementally after each modification according to the attribute grammar. Since attribute grammars are dependent on plans for attribute evaluation, the semantic grammar is preprocessed resulting in evaluation sequences. The evaluation sequences are interpreted during attribute evaluation.

Finally, the code generator components are generating code for a block at a time. The translation to intermediate code is based on interpretation of code generation grammars, which are variants of attribute grammars. Similarly as the semantic grammars, the code generation grammars are preprocessed before interpretation.

Editing grammar structures

Since all the grammars, or the aspects of a grammar, are represented in the same way as programs, they may be edited by the same structure editor. When editing a grammar, the editor is interpreting a grammar description of a grammar, i.e. a meta grammar. The meta grammars are also represented as abstract syntax trees and may be edited by the structure editor as well. Figure 7 shows the target program, the language grammars (or grammar aspects) and the meta grammars. It also shows the structure editors used for editing the language grammars. Notice that the structure editors in figure 7 consists of both an editor component interpreting abstract grammars and a presentation component interpreting concrete grammars.

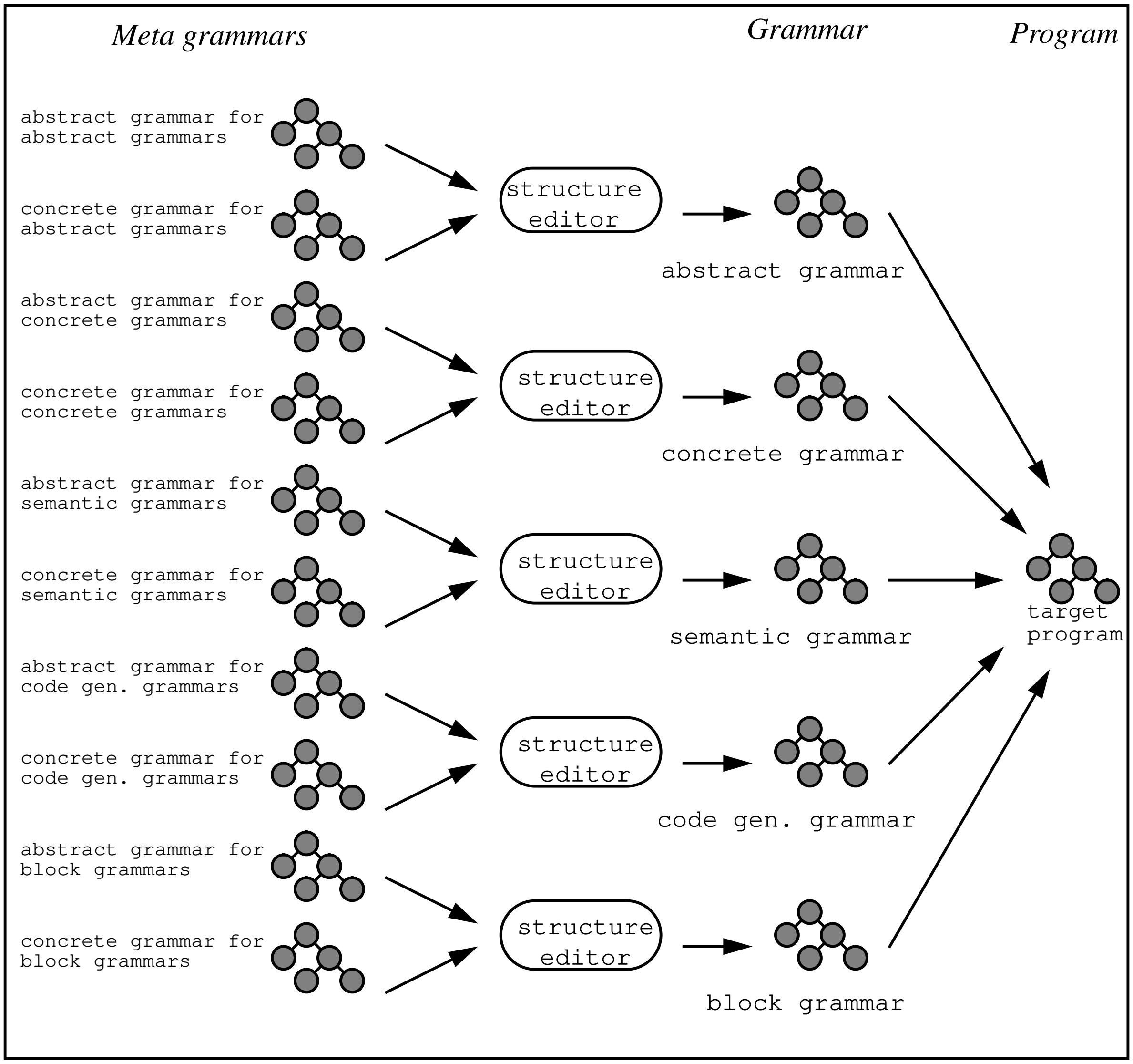

FIGURE 7. Meta grammars, grammars and target program

In figure 7, the target program structure is shown to the right. The target program is edited and executed by means of the Orm components, which are interpreting the language grammars, represented by arrows in the figure. Simultaneously as the language grammars are interpreted they may be edited by structure editors having the language grammar as target structure while interpreting meta grammars. For instance, the abstract grammar may be edited by a structure editor interpreting an abstract grammar describing the structure of abstract grammars and a concrete grammar describing the presentation of abstract grammars. In the same way any of the language grammars (or the grammar aspects) may be edited in accordance with the respective abstract and concrete meta grammars.

The figure is not symmetric, since the language grammars only are manipulated by the editor and presentation component according to the abstract and concrete grammars. In future versions of the environment we would like to include all grammar aspects also in the meta grammars resulting in five meta grammar aspects for each language grammar aspect. The block grammar could be used for browsing among productions. The semantic grammar could be used for checking static semantics in grammars, for instance if an attribute in an attribute grammar is declared in that production. The code generation grammar might be used for preprocessing in the attribute grammars generating plans for attribute evaluation.

The Meta Grammars

The meta grammars are defined and presented in terms of themselves. Figure 8 shows the relationships between the meta grammars, i.e. how meta grammars define and present each other.

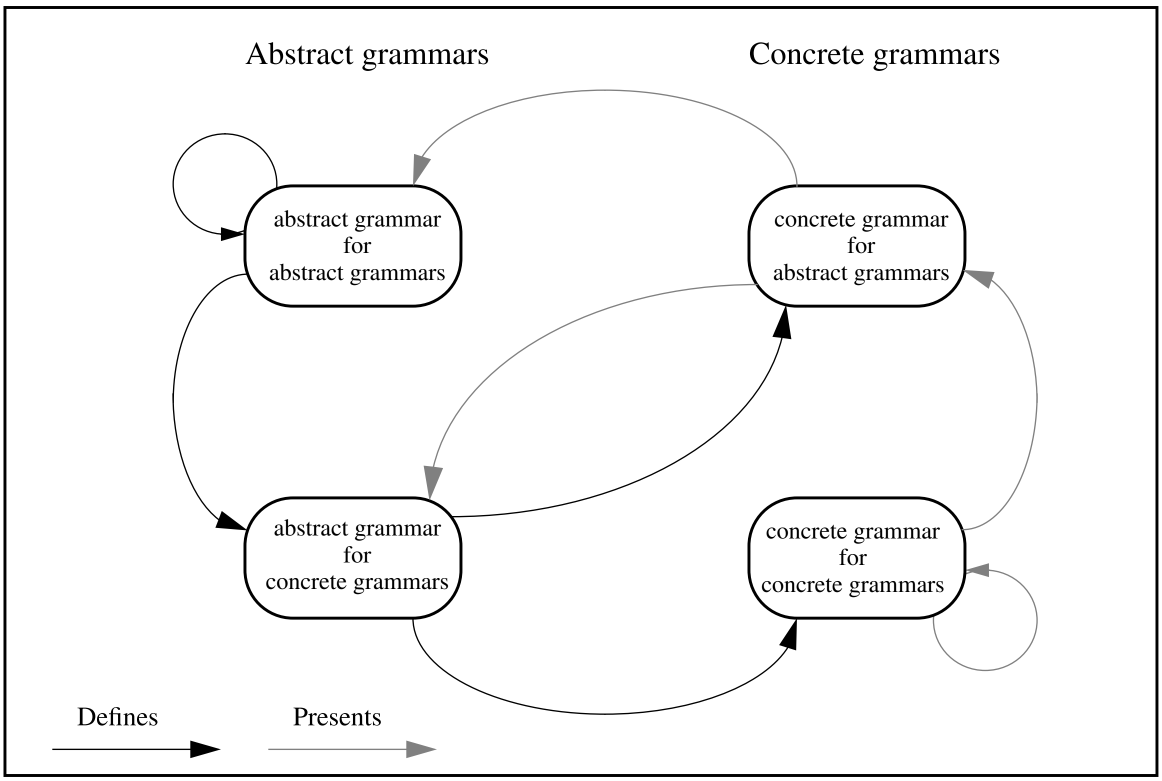

FIGURE 8. The relationship between abstract and concrete meta grammars

Since Orm can be used for editing any hierarchical structure defined by grammars, it is possible to edit any of the meta grammars. However, it is only sensible to edit some of them. The concrete grammars for any of the language aspects may be modified freely, which make it possible to modify the presentation of grammars and meta grammars. The abstract grammars may, however, not be modified at all. This is due to the fact that the structure of the grammars is defining the language understood by the Orm components.

They must not be modified without requiring corresponding modifications of the program components.

3.3 Using Orm for Language Design, Maintenance, and Customization

The fact that all the aspects of the language grammars are interpreted results in a high degree of flexibility for the user to modify both the structure and the surface syntax of a language as well as the static and dynamic semantics. In environments generated from language grammars, only the implementor, i.e. the person generating an instance of the environment from a grammar, has this ability. The end user of the generated environment has no ability to influence it.

Customizing surface syntax

By changing the concrete grammars of a programming language, the surface syntax may be changed completely. The potential of this ability should, however, not be overestimated. The concrete grammars in Orm are rather primitive, but even with more elaborate presentation schemes the possibilities are limited. A programming language is to a large extent dependent on the abstract structure of the language. Changing the concrete grammars can, however, be useful. Users often have different opinions about how a program should be formatted in order to gain in readability and aesthetic qualities. It is partly a matter of taste, but a familiar layout certainly makes a program easier to read and comprehend. This has lead companies to develop standards for program layout in order to support different programmers to read each others programs. Through the use of a system where the user may specify his or her own personal layout, such discussions become irrelevant. The layout is not a property of the individual program.Various presentations may easily be created without affecting the abstract representation of the program. The program layout is thus a matter of personalization. The personalization is defined interactively by the user himself, only limited by the expressive power of the concrete grammars. If the user wishes, the surface syntax may be totally redefined.

Command language interface

In [Minör 87], the use of the Orm structure editor as a command language interface is discussed. For command interaction it is reasonable to provide different surface syntax of the command language for different user categories. Beginners may prefer a more verbose dialog with explanations, whereas expert users prefer a more compact surface syntax. The different presentations of the command structures may be set up in Orm in several alternative concrete grammars for different levels of expertise. The language used in the presentation (Swedish or English for instance) or translation between languages used in the command interaction can also be done in a straightforward and well defined manner. Furthermore badly designed command languages with incomprehensible commands may be improved, at least at the surface level. The ability to personalize the command language of an interface interactively is, of course, desirable.

Text processing

Modifications of concrete grammars may be of use in text processing as well. The text contents, paragraphs, chapters, and different styles are represented in the abstract syntax tree according to the abstract grammar. The concrete grammar specifies the actual layout chosen. As an example consider reference lists. A reference typically is a structure with author(s), title, magazine or book or proceedings, volume, number, month, and year. The typography used in different reference lists varies although the contents is identical. Converting from one format to another may be a tedious task. An abstract representation of a document together with a presentation that is modifiable by the user interactively would be desirable. However, the example requires a concrete grammar more powerful than the one provided in Orm at present, with support for different fonts, italic, underline, etc.

Language design and maintenance

Language design is an area where the ability to modify the syntax and semantics interactively is interesting. Designing a programming language is a long term process where several design decisions have to be considered carefully. However, language design is not restricted to the design of traditional programming languages. In many cases application-oriented languages, or *little languages* [Bentley 86], are designed as an interface to an application, e.g. command languages, database query languages, specification languages, configuration languages, and grammar formalisms. The ability to create and modify grammars interactively supports rapid prototyping of languages. A modification of the grammar will directly come into effect and the result becomes visible for evaluation instantly. The importance of *iterative design* in general has been emphasized e.g. in [Brooks 87]. The interpretation of grammars supports iterative design of both the syntax and semantics of languages. The language designer is allowed to experiment with the language and to evaluate it repeatedly while it evolves.

A related area is software environment maintenance. Since most of the language specific information is expressed in grammars in Orm, a large deal of the language-related maintenance in traditional environments becomes grammar maintenance in Orm. Like programs, grammars are not correct from the beginning. In Orm grammar errors may be corrected and the result may be evaluated without the need of a time-consuming regeneration of the environment. Grammars may be “debugged” interactively. The environment also makes it easy to adapt a grammar to a new version of a language or to a specific dialect.

A general problem when having a language description and an instantiation of it, e.g. a program, is the inconsistency of the program and the new description after modifying the description. The problem applies to software environments in general, but it is particularly evident in grammar-based environments. The grammar interpretive technique itself is not causing the inconsistencies. Systems generating environments from grammars are subjected to the same problem. However, the ability to modify grammars interactively encourage frequent modifications and hence potentially more inconsistencies. Inconsistencies resulting from changes of concrete, semantic and code-generation grammars are easy to deal with since the screen representation, the semantic attributes, and the code can easily be regenerated even though it may take some time. The abstract grammar, however, defining the structure of the representation is a harder problem.

A rather simple approach to deal with inconsistencies is used in DOSE [Feiler et al. 86]. Target structures inconsistent with the grammar are simply discarded. The approach is, of course, not satisfactory, even though it actually results in consistent grammars and programs. More elaborate approaches based on semi-automatic transformations have been suggested in [Nørmark 87] and [Garlan et al. 87]. None of the approaches allow target structures to be transformed automatically after complex grammar modifications. Orm follows a different strategy. Inconsistent programs have to be transformed "manually" after any modification of the abstract grammar causing inconsistencies. However, not all modifications do result in inconsistencies. An abstract grammar can be extended, for instance, without causing inconsistencies.

4 Orm in Practical Use

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 to serve as a basis for the integration of more advanced features needed for programming in the large and covering a broad range of the software life cycle. An alternative would have been to stick to 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.

Orm is implemented in Simula and runs on Sun SPARCStations. As a programming environment it is in the test state and has been used for implementation of small programs (up to 1000 lines or so). As a language laboratory, the system has proven highly attractive although it still has some "rough edges". The largest grammar implemented is the one for Simula, consisting of about 100 productions in each grammar aspect. Printed out, the grammar is about 4000 lines of sparse text. In handling full programming languages there are currently shortcomings, mainly due to some semantic functions not being general enough. For this reason it has not yet been possible to implement for instance a full Beta system.

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 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.

The grammar editing facilities has been valuable within the Mjølner project. It has been used for developing the language grammars for Simula and partly Beta. All the grammar aspects for these languages have been developed and “debugged” interactively.

The grammar formalisms for the block, abstract, concrete, semantic, and code generation grammars have also been developed using Orm. This has been done by developing the meta grammars, i.e. the abstract and concrete grammars, for each grammar. The first versions of the meta grammars for abstract and concrete grammars were, however, initially entered using other tools in order to bootstrap the grammars. The meta grammars for the block, semantic and code generation grammars have been developed using Orm exclusively. The grammars have evolved successively and modifications have been frequent. Orm has shown to be a practicable tool for experimenting with the formalisms.

Outside the project, Orm has been used as a meta environment in teaching the compiler construction course at our department for several years with good experience. The students have been able to experiment with grammars and languages, and it has been possible to evaluate the results of grammar modifications immediately. Several companies have used Orm in the same way for prototyping language syntax.

Orm also has been used for “little” languages. At a company, grammars for the Unix *make* language were developed. Make is known to have a cryptic syntax which is hard to learn and use. The support offered by the structure editor have shown to be useful. An alternative more verbose and readable concrete syntax was defined. The alternative concrete syntax was shown to users while the original concrete syntax was used when the makefile “program” was sent to make for execution.

Orm has been use by Swedish telecom companies for language prototyping in several projects. The ability to obtain a structure editor in short time and to evaluate and refine the language constructs has been reported to be very useful. It was also used to demonstrate ideas of a prototype language to a larger group of users. The users were given Orm with the prototype grammar and were allowed to play around with the language in order to evaluate it and give feedback. Based on the feedback, the language design was refined. The structure editing was considered to be very useful when exposing users to a new language design, since the users did not have to be experts in the syntax. The structure editor support allowed them to construct e.g. programs or data base schemes for evaluation without prior training.

Structure editing in general has shown useful when using Orm as a meta environment. Since the grammar formalisms used are not well-known, which is often the case with programming languages, the support of the structure editor has been appreciated. Even among the people who have designed the notations themselves, the support was needed. One reason for this is that the users are “occasional users”, i.e. they do not use the notations daily and after a month or two they needed support when editing a grammar. Another reason is that the notations sometimes have changed from day to day, particularly in the initial phases of the project. The structure editors have helped in presenting the up-to-date version of the grammar notation.

Orm has been used in practice as a programming environment in teaching the first programming course at the department. A group of 15 students used the Orm environment when the rest of the students used a traditional compiler/debugger. In the course, object-oriented programming was taught using Simula. In large, the experiences using Orm were good. There was no examination in the end of the course making it hard to evaluate if the student using Orm made better or worse than the other students. A fairly comprehensive questionnaire was filled in by the students using Orm making it possible to make a rough evaluation. The questionnaire was handed out after the students had used the traditional Simula system in the next course enabling them to compare Orm and a conventional environment. Most students had additional experience from using other systems from outside the university.

Most complaints about Orm concerned execution speed. The system had in the course to be used on too small and too slow machines to give Orm a fair chance as an interactive system. Despite the bad response times, the students were surprisingly positive. 80% of the students considered Orm to be better or much better than a traditional environment. All students thought the system was fairly easy to learn. Over 80% felt that they had control over what happened in the environment. The integrated incremental nature of the environment was considered to be of great value by all students. Almost all students had “fun” using the system.

Concerning structure editing, no student was unhappy with constructing programs by means of menus. When asked whether they preferred using menus or editing program fragments as text, 20% answered that they used both to equal degree and 80% preferred using the menus. All students considered the context-sensitive editing technique to be very useful and 93% of the students preferred using this mode to entering identifiers as text. Furthermore, the hierarchical browser was received positively by all students. On the question whether they had preferred to use a conventional text editor instead of the structure editor, 93% answered “no” or “absolutely not” (7% answered “maybe”). On the questions about other parts of the environment, the students were almost exclusively positive.

Finally, a surprising observation about programming style was made at the programming of assignments. Several students using Orm placed the declarations of variables at the wrong block level. Usually the variables were declared globally instead of locally. Apparently, the Orm browser makes the (geometrical) distance between blocks smaller: a global declaration is done as easy as a local one. The browser did not, as it was intended to, convey the hierarchical scopes of the language to the extent that such mistakes were avoided. None of the students using the conventional system did that kind of errors.

Besides the use of Orm reported above, the system has been distributed to other universities and research labs for evaluation and experimenting. At the time of writing the system has been distributed, to some 30 sites. We hope to get feedback on the system and gain experience to improve it.

5 Future Research

Orm has been developed to a fairly complete and working prototype. However, it is still far from a system that can be used in “real” applications, e.g. in industrial settings. But the system may be used as a platform for further research concerning both the language and software development support.

Concerning language development, we would like to extend the functionality and the meta grammars to give more support for browsing in grammars and check semantic consistency, as described in section 3.3. We also would like to give more support for language “debugging”, e.g the possibility to follow the attribute evaluation in the semantic and code generation grammars. Better support for transformations of abstract syntax trees between different versions of the grammar is also needed.

The support for “little languages” is an area interesting to explore further. The integration of the Orm meta environment to different environments using the little language as interface has to be further developed. Another area in this context is grammar modularization. When developing different little languages, as well as different programming languages, a general problem is that parts of different languages are identical or similar, e.g. the syntax and semantics for arithmetic and boolean expressions. The ability to share grammars or grammar parts between different languages would be a solution to this problem. In the same manner as code is shared and reused by using modules and separate compilation in software environments, grammars could be reused. In a system supporting *grammar reuse*, it would be possible to define “production libraries” from which the syntax and semantics of different language constructs could be imported and further specialized.

Concerning software development, semantic support is an area needing further investigation. The context-sensitive editing in Orm has shown to be very useful. The idea to use the semantic information stored in the abstract syntax trees for supporting the user to read, understand, and browse in a software system is a more powerful way than available in conventional text-based systems. By improving this kind of facilities, the power for giving valuable language-based support in structure-based environments could be demonstrated.

Programming-in-the-many, or cooperative software development, is an interesting and demanding area for future research. Software development is frequently teamwork, but commonly used systems are weak in supporting groups working together. Orm is at present a single-user environment, even though its revision control system together with its module handling gives some support for developing software and grammars in a project. We believe the environment is a suitable platform for developing an environment supporting multiple cooperating users. This believe is based on the following properties of the environment:

- The structure-oriented architecture
- The integrated configuration control system
- The graphical browser

First, the structure-oriented architecture facilitate full control in the system. All information about a program is collected in one representation, which facilitates sharing of the information among multiple users. If some part of a program is changed by one user, it is possible for any tool to keep track of its consequences and notify other users if desired. Second, the integrated configuration control system keeps track of versions and revisions. It also handles alternative development lines of a program, which is crucial when developing software in a project. Finally, the graphical browser is used for navigation in the program by means of a hierarchy of windows reflecting the program structure. Since the presentation is structured, we believe it will facilitate visualization of several users working on one software component.

6 Summary

We have presented Orm, a structure-oriented 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 developed further in Orm. Orm also differs from these environments 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 tested on a target program. This applies not only to developing programming languages, but to developing any formal language, including the meta formalisms 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 traditional text-based environments.

Acknowledgments

The Orm environment has been developed within the Department of Computer Science at Lund University. Several people have been involved in the work described in this paper. Mats Bengtsson has been working with the run-time environment and garbage collection, Lars-Ove Dahlin has worked with incremental semantic checking together with Görel Hedin who also designed the attribute grammar formalism, Göran Fries has been working

with the back-end code generation generator, Anders Gustavsson has designed the configuration control system, Dan Oscarsson has supplied the hierarchical window system, Magnus Taube made the run-time observation tool, Sten Minör has been involved in the structure editor, the grammar interpretation, and the front-end code generator, and finally Boris Magnusson has been project leader of the Orm subproject within the Mjølner project.

The work reported in this paper has been carried out as part of the Nordic Mjølner project. The project is supported by The Nordic Fund for Industrial Development and The Swedish National Fund for Technical Development (STU - NUTEK).

References

[Bahlke et al. 92] Bahlke, R., Snelting, G., *Design and Structure of a Semantic-Based Programming Environment*, to be published in International Journal of Man-Machine Studies, Academic Press, 1992

[Bengtsson 90] Bengtsson, M., *Real-Time Compacting Garbage Collection Algorithms*, licentiate 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

[Brooks 87] Brooks,F.P.Jr., *No Silver Bullet - Essence and Accidents of Software Engineering*, IEEE Computer, April, 1987

[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

[Fritzson 84] Fritzson, P., *Towards a Distributed Programming Environment Based on Incremental Compilation*, Ph.D. thesis, Dissertation no. 109, Department of Computer and Information Science, Linköping University, 1984

[Garlan et al. 87] Garlan, D., Krueger, C.W., Staudt, B.J., *A Structural Approach to the Maintenance of Structure-Oriented Environments*, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, Vol. 22, No. 1, 1987

[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

[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 91] Hedin, G. *Context-Sensitive Editing in Orm*, Dept. of Computer Science, Lund University, 1991,

[Hedin 92] Hedin, G., *Incremental Semantic Analysis*, Ph.D. thesis, Dept. of Computer Science, Lund University, 1992

[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

[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

[Madsen et al. 90] Madsen, O.L., Magnusson, B., Møller-Pedersen, B., *Strong Typing of Object-Oriented Languages Revisited*, Proceedings of OOPSLA ‘90 Conference on Object-Oriented Systems, Languages, and Applications, 1990

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

[Medina-Mora 82] Medina-Mora, R., *Syntax-Directed Editing: Towards Integrated Programming Environments*, PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1982

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

[Minör 87] 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 90] Minör, S., *On Structure-Oriented Editing*, PhD thesis, Dept. of Computer Science, Lund University, 1990

[Minör 92] Minör, S., *Interacting with Structure-Oriented Editors*, to be published in International Journal of Man-Machine Studies, Academic Press, 1992

[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

[Nørmark 87] Nørmark, K., *Transformations and Abstract Presentations in a Language Development Environment*, PhD thesis, Computer Science Department, Aarhus University, 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

[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

[Teitelbaum et al. 81] Teitelbaum, T., Reps, T., *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment*, Communications of the ACM, Vol. 24, No. 9, September, 1981

[VanDeVanter et al. 92] Van De Vanter, M.L., Ballance, R.A., Graham, S.L., *Coherent User Interfaces for Language-Based Editing Systems*, to be published in International Journal of Man-Machine Studies, Academic Press, 1992

[Winograd 87] Winograd, T. A., *Muir: A Tool for Language Design*, Report No. STAN-CS-87-1159, Department of Computer Science, Stanford University, 1987