# INCREMENTAL SYMBOL PROCESSING

by

Peter Fritzson

Department of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden
EMAIL: paf@ida.liu.se

*Abstract:*

This paper introduces a novel entity-relational model for incremental symbol processing. This model forms the basis for the generation of efficient symbol processing mechanisms from high-level declarative specifications and query expressions, using program transformation techniques such as data type refinement.

The model is conceptually simple, but powerful enough to model languages of the complexity of Ada. The new model is compared to earlier, more restricted, incremental hierarchical symbol table models. The differences between symbol processing in conventional compilers and incremental symbol processing are also discussed.

## 1. INTRODUCTION

Symbol processing and scope analysis is one of the tasks usually performed by the syntax and semantics analysis phases of compilers. As programs grow larger, containing more interfaces and declarative information, symbol processing operations such as definition and lookup tend to consume a large fraction of the total compilation time. Also, scope analysis and lookup of definitions can be quite complex for several languages in the ALGOL family, where ADA is one example.

It is well known that interactive and incremental programming environments can enhance the programming process by preventing or quickly detecting errors, and by helping the programmer maintain and understand large programs. Thus it is essential that the programming environments use incremental methods for symbol processing.

This paper introduces a high-level declarative entity-relational data model for incremental symbol processing and scope analysis. The new model, which also can be regarded as

---

*This research was supported by the National Swedish Board for Technical Development*

object-oriented, gives several advantages:

Language-independence
: The model contains a few simple, yet powerful, language-independent modelling primitives. Thus it serves well as the basis for generation of language-oriented incremental symbol processing mechanisms from compact specifications.

Integration
: Since the model is entity-relational, it can be interfaced to existing relational database technology. Thus the symbol table can be viewed as a part of an integrated relational program database, which can hold both programmatic and documentation information.

Query language
: The relational formalism provides a general query language.

Efficient compilation of Queries
: The current model has been embedded into a very high level language which includes program transformation facilities [Refine-87]. Using such transformations the current model can be compiled to lower level procedural code for in-core data bases, ultimately matching the efficiency of handwritten symbol table packages.

The current model is aimed at languages such as Pascal, Modula-2, C, Ada, etc., but may well have wider applicability. An earlier incremental hierarchical symbol table model for Pascal is presented in this paper as a comparison. As mentioned, the model can be viewed either as entity-relational or object-oriented, it is largely a matter of choice. Entities are objects. A disadvantage of a pure object-oriented view is the absence of a general query language. A disadvantage of a pure relational implementation is that the performance of current relational databases is not satisfactory for compiler symbol processing applications, see e.g. [Linton-84]. Therefore, the methods presented here more or less assume an in-core database, or a database with a high degree of clustering and in-core caching of relevant data.

There are at least two alternatives of integrating this model into a compiler or editing environment. Through the fundamental operations Define and Lookup represented as procedures, the model can be interfaced to an incremental compiler [Fritzson-83] or to an editing environment based on action routines [Medina-Mora,Feiler-81]. Alternatively, Define and Lookup can be regarded as implicit relations, and the model interfaced to an editing environment based on relationally attributed grammars [Horwitz,Teitelbaum-86].

Attribute grammars are currently a popular means of specifying semantics for programming languages. Their strength is a declarative equational style notation, which enhances correctness and readability. Another advantage is the existence of general incremental attribute propagation methods for editing applications [Reps-83]. However, there are also disadvantages. The attribute grammar style of specification is somewhat low-level - each equation specifies a too small fraction of the total computation. This fragmentation can make the attribute grammar formulation of certain problems to be hard to understand. For example, the specification of syntax and semantics of Ada requires a 20000 line attribute grammar [Uhl,et.al-82]. Also, although advances have been made, efficiency still seems to be a problem. Difficulty in optimization is often due to information loss and constraints introduced when a problem is expressed in a too low-level formalism. For example,

optimizers of intermediate code often need to reconstruct control-flow and data-flow which may have been explicit in higher level formalisms. Another example is the copy bypass optimization [Hoover-86], which eliminates unnecessary copy operations introduced by the constraints of attribute grammar formalisms.

We propose an alternative paradigm, based on program transformations, in the search for suitable specification languages. In this paradigm, the freedom in choosing transformations should create a greater chance of combining clarity of specification languages and efficient execution of target code. On the one hand, specialized high level notations can be devised for certain application areas. Certain notations are powerful precisely because they have a narrow applicability - more can be expressed with less. On the other hand, there are also general formalisms with powerful constructs. Examples are relational and set-theoretic operators, pattern-matching and logic. Such very high level notations can be compiled into lower level efficiently executable code by program transformation techniques such as automatic data structure refinement and control structure refinement, see [Goldberg,Kotik-83]. A special case of data structure refinement can be found in [Horwitz,Teitelbaum-86] where queries on implicit relations are transformed into queries on tree structures. The very high level specification-style notation implies greater freedom in applying various optimizations. Our symbol processing model is aimed at being a step in this direction, oriented towards the task of generating incremental programming environments from specifications.

Since the transformations from specification to executable code are automatic, correctness of executable programs follows automatically if the transformations have been proved correct, and if the specification is correct with respect to intuitions. Correctness criteria for transformations are briefly described later in this paper.

## 2. BACKGROUND

Programs define and use programming objects such as variables, types, and procedures. These objects are referred to in programs by means of symbols. The symbol processing component of a programming system supports all activities that define and use symbols referring to programming objects. These activities include the use of symbols by tools such as the compiler, linker, debugger, and librarian, and the browsing of symbols by users in cross reference queries.

By a *symbol table* or *symbol database* we mean the totality of the state needed to manage the definition and use of the symbols required for producing a software object, e.g., an executable module, from source program units. This state can be described as a set of relations between a set of entities. For example, there may be a relation *define* that describes all definitions within a set of source modules. It would relate entities such as the symbols being defined, the modules in which they are defined, and the declaration objects the symbols represent. A symbol processing system provides a programmatic interface to the state described in the symbol table.

A symbol processing system is *integrated* if the same symbol table is used for all programmatic and user activities, e.g., compiling, linking, debugging, cross referencing, and browsing. An integrated symbol processing system has the advantage that information is not duplicated in several places with possibly inconsistent format. A change in a source program can be incorporated once into the symbol table, and used by all applications. The system provides a uniform interface to the symbol table information, so that applications using this information are easier to write and maintain. A symbol processing system is *incremental* if the symbol table is persistent, and if a change in the source code dynamically causes corresponding updates in the symbol table. An incremental symbol table is one important prerequisite to make it possible for small changes to source code to be incorporated at minimal cost into corresponding executable program, object code modules, and symbol table database.

Before we continue our discussion on the role of incremental symbol processing in incremental compilation, let us briefly discuss the difference between separate compilation and incremental compilation. In both cases the goal is to decrease the turn-around time by re-using results from previous compilations. However, there is an order of magnitude time difference between these two technologies. In program development environments based on incremental compilation the time lag until execution can be resumed after a small program modification is usually a few seconds or less - even for big programs. By contrast, the recompilation and relink time in traditional separate compilation environments is usually measured in minutes or more, even though advanced change analysis [Tichy-86] has improved the situation. In addition the program need to be restarted from scratch after a rebuild.

The better performance of incremental compilation technology depends on several factors. The unit of recompilation is smaller for incremental compilation - a procedure or statement - than for separate compilation where it is usually a file or module. Also, maintenance of dependencies between declaration objects is at a finer granularity in an incremental environment. Incremental system components such as compiler, debugger and linker are better integrated - sometimes they even do not exist as separate entities! For example, in traditional environments, the compiler, debugger and linker usually build their own symbol tables, whereas a true incremental environment uses the same symbol table for all these purposes. In addition, an incremental environment usually preserves the current execution state, which makes debugging more convenient by providing continued execution after most small program changes.

## 3. INCREMENTAL SYMBOL PROCESSING IN INCREMENTAL COMPILATION

Incremental symbol processing is one important part of the total incremental compilation transformation, which translates source code and old executable code to updated executable code. Symbol processing becomes increasingly important when compiling really big programs [Rational-85], [Conradi,Wanvik-85], since there is an increased amount of declarative context in the form of include files and module specifications.

The sub-transformations which comprise incremental compilation are shown very schematically below. Note that all input arguments are not explicitly shown. For example, incremental code generation takes both an old version of the object code and an abstract syntax tree as input, and produces a new version of the object code. This means that the arrow ---> is not just a mapping, it is an update transformation.

Note also that incremental parsing is not needed if the program is stored in tree-form, and that incremental optimization is optional. In some systems incremental semantic analysis also includes incremental symbol processing.

```
Incremental Compilation:          Source code              --->  Executable code
```

The total incremental compilation transformation above can be decomposed into:

```
Incremental Code Generation:      Abstract syntax tree  --->  Object code
Incremental Symbol Processing:    Declarations          --->  Symbol table
Incremental Linking:              Object code           --->  Executable code
Incremental Execution:            Execution state       --->  Execution state

Incremental Parsing:              Source text           --->  Syntax tree
Incremental Semantic Analysis     Attributed tree       --->  Attributed tree
Incremental Optimization:         Intermediate code     --->  Intermediate code
```

In the rest of this paper will concentrate on incremental symbol processing, and not be concerned with other parts of incremental compilation.

## 4. BASIC SYMBOL PROCESSING OPERATIONS

There are two basic operations in symbol processing, *definition* and *lookup*.

The *definition* operation adds a new definition of a symbol to the symbol table. This involves adding the symbol together with a description of the declaration object it will henceforth represent. Thus a mapping is established from a symbol in some context to a declaration object. Note that a declaration object is analogous to a symbol table entry in conventional compilers.

The *lookup* operation. Given a symbol that is being used in a context somewhere in a program, find the declaration object which this symbol refers to.

In addition to the two basic operations common to all symbol processing mechanisms, an incremental symbol processing mechanism must support *insertion*, *deletion* and *update* of declaration objects. It must also incrementally maintain *dependencies* between declarations, and support navigation operations which change the current context.

# 5. INCREMENTAL SYMBOL PROCESSING VERSUS CONVENTIONAL SYMBOL PROCESSING

## *5.1 Dependency maintenance and update of declarations*

An incremental symbol processing system needs to support efficient access and updating of dependencies between declaration objects. This is especially important in order to incrementally support updating of global declarations.

The definition of a declaration object can be incrementally inserted, changed or deleted from a program. For example, after a change to an existing declaration object, e.g. a global type declaration, the system has to re-elaborate all objects which are dependent on this declaration. Thus, the system needs efficient access to global dependencies in order to quickly find the dependent objects. If a new declaration object is inserted, the system also has to determine how this affects the visibility of existing objects, e.g. if it will hide declarations in enclosing blocks.

## *5.2 Sequential versus Random Access*

The symbol processing unit of a conventional compiler usually makes a sequential pass over a source program unit, e.g. a file, and performs actions of defining or looking up symbols as they are encountered during this sequential pass. This means that at any given point, the current state represented in the symbol table only reflects definitions which have so far been encountered before the current symbol. A conventional compiler also has schemes for remembering and updating the current block and the visibility of symbols as it progresses sequentially through a source program.

An incremental symbol processing system, on the other hand, has to be able to make "random access" to just those parts of a source program unit that have been changed. It cannot rely on assumptions based on the sequential processing of source programs. Any such assumptions that affect scope and visibility have to be explicitly represented in the symbol table. The incremental symbol table needs a way of keeping track of the relative positions of symbols in a source program unit. Many programming languages are compiled by single-pass compilers and thus require that a definition occur before its use. For such languages this positional information is of critical importance for determining the meaning of a program.

# 6. SYMBOL PROCESSING MODELS

In the following sections we will present three incremental symbol processing models - a *hierarchical model*, an *attributed abstract syntax tree model*, and an *entity-relational* symbol processing model. The first two are just discussed briefly. The emphasis is on the entity-relational model, which is higher-level and and declarative.

## 7. A HIERARCHICAL SYMBOL PROCESSING MODEL

We briefly present a simple hierarchical symbol processing mechanism for Pascal. The symbol table consists of a tree of local symbol tables, which allows both insertions and deletions of declarations. This incremental symbol table model has been implemented in the DICE system [Fritzson-83], [Fritzson-85], which is a programming environment based on incremental compilation. Note that the symbol processing mechanism presented here does not represent declaration position within a sequence of declarations. Thus, it allows forward referencing of declarations which is consistent with Modula-2, but without enforcing the Pascal define before use semantics.

A simple program example about drawing boxes will illustrate symbol processing operations in several different models, and also the structure of these models. The concept of region, defined in the next section, is also exemplified.

```
PROGRAM BoxProg;

TYPE  BoxType = RECORD
                  X1, Y1 : Integer;   X2, Y2 : Integer;
                END;
      Color   = (Red, Green, Blue);

VAR
      BoxVar : BoxType;


PROCEDURE DrawBox(X1,Y1,X2,Y2 : Integer);
  VAR
    Dx, Dy : Integer;

  PROCEDURE DrawLine(DeltaX, DeltaY : Integer);
    BEGIN (* DrawLine *)
      .....
    END; (* of DrawLine *)

  BEGIN (* DrawBox *)
    GotoXY(X1, Y1);
    Dx := X2-X1;
    Dy := Y2-Y1;
    Drawline(0, Dy);
    Drawline(Dx, 0);
    Drawline(0, -Dy);
    Drawline(-Dx, 0);
  END; (* of DrawBox *)

BEGIN (* BoxProg)
  ClrScreen;
  .....
  WITH Boxvar DO BEGIN
    DrawBox(X1, Y1, X2, Y2);
  END;
END. (* of Main program BoxProg *)
```

*Figure 1. A program example BoxProg.*

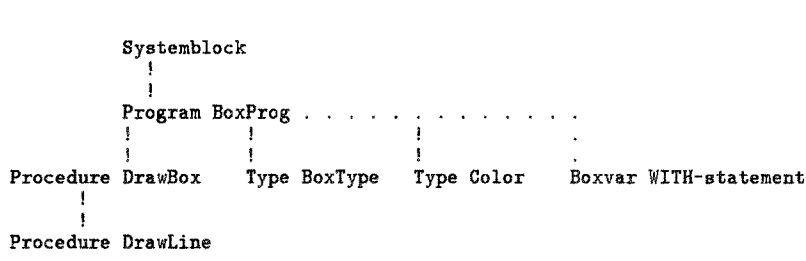


*Figure 2. Slightly simplified Region hierarchy.*

```
Level 0:
            Localsymtab [GotoXY, ClrScreen, BoxProg]
            Dependencies [(GotoXY -> BoxProg), (GotoXY -> DrawBox)
                          (ClrScreen -> BoxProg)];

Level 1:  Program BoxProg:
            LocalSymtab [Boxtype, Boxvar, DrawBox, Color, Red, Blue, Green]
            Dependencies [(Boxtype -> Boxvar), (Boxvar,DrawBox -> BoxProg)]

Level 2:  Type Boxtype:
            LocalSymtab [X1,Y1, X2, Y2]
            Dependencies [(X1 -> BoxProg), (Y1 -> BoxProg),
                          (X2 -> BoxProg), (Y2 -> BoxProg)]
Level 2:  Type Color:
            LocalSymtab [Red, Blue, Green]
            Dependencies [ ]

Level 2:  Procedure DrawBox
            LocalSymtab  [X1, Y1, X2, Y2, Dx, Dy, Drawline]
            Dependencies [(Drawline -> DrawBox),
                          (X1,Y1,X2,Y2,Dx,Dy -> DrawBox)]

Level 3:  Procedure Drawline
            LocalSymtab  [DeltaX, DeltaY]
            Dependencies [ ]
```

*Figure 3. Contents of a hierarchical symbol table which includes dependencies.*

The hierarchy of local symbol tables and dependencies for the small BoxProg program is shown in Figure 3. Note that only dependencies between declarations and between declarations and procedures/functions are shown. For example, (GotoXY -> DrawBox) means that DrawBox depends on and uses GotoXY. Dependencies between a set of local declarations and the procedure/program body at the same level are represented using a special form, e.g. (X1,Y1,X2,Y2,Dx,Dy -> Drawline).

Note also that the WITH-statement in the body of program BoxProg is shown connected to the block/region hierarchy with a dotted line. This means that the local symbol table of field names associated with the record type Boxtype is temporarily associated with the block hierarchy when the current position is within the WITH- statement in the BoxProg program.

## 8. THE NOTIONS OF REGION AND SCOPE

The term *region*, or more precisely *declarative region* [ADA-83] , is sometimes used interchangeably with the term *block*. However, the notion of region is more general - it denotes any syntactically defined portion of a program, such as e.g. a record declaration.

The notion of block is usually connected with objects which instantiate some kind of activation records, e.g. procedure blocks or program blocks. Each pair of regions are either disjoint portions of a program, or one region is completely enclosed within the other.

The *scope* of a declaration object consists of the parts of a program where it is legal to reference the declaration [ADA-83]. Scope need not conform to the nice nesting structure of certain programming languages. For example, in the C language [Harbison,Steele-84], the scope of an identifier extends from its first occurrence - see example in Appendix A. Similar rules apply for Ada and Pascal. Scope can be complicated, since it depends on both position, region structure and the nesting rules of the languge.

## 9. OPERATIONS ON THE HIERARCHICAL INCREMENTAL SYMBOL TABLE

In addition to manipulating the contents of the symbol table, certain symbol processing operations support navigation operations, i.e. setting the current focus in the tree-structured symbol table. This focus can be represented by a cursor, which we call the *symcursor* in the context of symbol processing. In [Fritzson-85] two similar kinds of cursors are defined: the *editcursor* which denotes the context of editing and incremental compilation operations; the *execursor* which denotes the current context during debugging and execution.

*Navigational operations*

To *enter a block or region*, set the symcursor to the block which is to be entered. This has the effect that subsequent definition operations will as a default be performed on the local symbol table of this block. Lookup operations will start searching in the current local symbol table. The block to be entered has to exist - it must have been previously created during some define operation. The *leave block* operation is a special case of *enter block*. It means that the symcursor should be reset from denoting the current block to instead denote the parent block, if such a block exists.

*Define symbol*

This operation associates a symbol in the current context with a new declaration object. First, create a declaration object and elaborate type information specified by the declaration which defines the symbol. For certain declaration objects such as procedures or records, a block with a local symbol table is also created. Then do a partial lookup to check if the symbol is already defined in the current block, in which case the new definition is illegal. Finally, enter the symbol and its declaration object in the local symbol table.

*Lookup symbol*

Given a symbol used to reference an object, this operation finds the corresponding object. The lookup operation can be a complex function of scope and visibility rules together with current context, position and type information. However, in the special case of our hierarchical symbol table for Pascal it is simple: first perform a lookup in the local symbol table of the current block; while not found continue the lookup in subsequent parent blocks until the outermost system block has been searched.

*Declaration Update*

The *insert* operation is essentially a define operation. However in an incremental symbol table more tasks need to be done. For example, it must be checked if the new definition partially hides an existing declaration which is referenced in the current context. In such a case, all objects in the current context which are dependent on the previous declaration have to be re-elaborated and recompiled.

The *delete* declaration operation removes the symbol and its associated declaration object from the symbol table. Also mark all dependent objects for incremental recompilation. If the deleted symbol represents a variable, also free its associated target memory.

*Dependency Maintenance*

For each use of a declaration object, update the dependency structure to reflect this use. When a use of an object is deleted, then update the dependency structure to possibly remove a dependency.

*Dependency Query*

This query answers the question: *Who depends on me?* It returns the set of all declaration objects which are dependent on a certain object.

## 10. AN ATTRIBUTED ABSTRACT SYNTAX TREE MODEL

This symbol processing model is really a special case of the hierarchical incremental symbol table model described above. We note that the region hierarchy of the symbol table corresponds to the block hierarchy of the abstract syntax tree itself, so why not use the abstract syntax tree? We need not create a special local symbol table at each block level. Instead, it is possible to do lookup operations by searching declaration nodes in the current block in the tree. Elaborated declaration objects and dependencies can be attached to the tree as attributes. Insertions and deletions are performed on the attributed tree itself. This model is used e.g. by the Rational incremental Ada environment [Rational-85].

However, there are also disadvantages in representing the symbol table as an abstract syntax tree. It is less efficient to perform a lookup as a linear search through the tree instead of a single access to a hash table. Therefore the Rational incremental ADA environment has been augmented with hashed lookup at the global level [Rational-86]. Also, the implementation of symbol processing operations may become language dependent to a greater degree, since the tree structure of declarations is peculiar for each language. This model is of course not suitable for programming environments which do not use a tree structure as the primary program representation.

## 11. THE ENTITY-RELATIONAL DATA MODEL

The entity-relational data model was first introduced in [Chen-76]. It can be regarded as a thin layer on top of the relational database model [Codd-70]. It is convenient to think of a

relation as a table, or as a set of tuples $<v_1, v_2, .. v_n>$, where each data value $v_i$ belongs to a data domain $D_i$. The columns of the table are often called attributes. In the entity-relational model, there are two kinds of attributes: associations (between entities), and properties, see Figure 4.

| Concept | Informal definition | Examples |
|---|---|---|
| ENTITY | A distinguishable object (of some particular type) | Declaration object, (= symbol table entry in conventional systems), a node in an abstract syntax tree |
| PROPERTY | A piece of information that describes an entity | The name of a declared object such as variable. The memory size occupied by a variable. |
| ASSOCIATION (ATTRIBUTE) | A many-to-many or many-to-one relationship among entities. | CONTAINED-WITHIN: A variable X is declared within the declarative region of procedure FOO. |
| SUBTYPE (SUBCLASS) | Entity type Y is a subtype of entity type X if and only if every Y is necessarily an X. | The subtype EXPRESSION is a subtype of TREENODE. Subtype DECLOBJECT is a subtype of PROGRAMOBJECT. |

*Figure 4. The Entity-Relational data model*

A query is an expression in the relational algebra in which relational operators are applied to argument relations to produce a relation as a result. The three special relational operators are JOIN, SELECT and PROJECT. PROJECT is a unary operator that forms a new relation consisting of a subset of the attributes of a relation (or of the columns of a table). JOIN is a binary operator: it merges tuples from two argument relations together into bigger tuples, but selecting only those merged tuples that fulfil a given condition. SELECT is a unary operator that selects the subset of tuples in its operand relations that satisfy a given condition.

A *view relation* is a relation which is computed from other relations when needed, using an expression consisting of relational operators.

An *implicit relation* is never constructed. Instead it is implicitly represented by some other data structure. An example is the ANCESTOR relation which can be implicitly defined by a tree, but need not be constructed explicitly. Instead, the relational query operators are transformed into different query operators that operate on this other data structure. Such an example can be found in [Horwitz,Teitelbaum-86] where it is described how certain queries of implicit relations can be transformed into a equivalent queries on tree structures. This is a special case of data structure refinement [Goldberg,Kotik-83]. The DEFINE and LOOKUP operations mentioned previously can be represented as implicit relations in our model.

## 12. AN ENTITY-RELATIONAL SYMBOL PROCESSING MODEL

We have chosen to express our symbol processing model as an entity-relational data model [Chen-76]. This combines most advantages of the relational approach and the object oriented approach [Zdonik,Wegner-85], [Birtwistle,et.al-73]. The relational approach provides the powerful relational operators and a query language. The object-oriented approach provides subclassing with inheritance, in addition to association of attributes and operations with objects.

There exist certain limitations of the relational model. The relational operators cannot handle (1) queries that require transitive closure, (2) queries that require order-dependent processing, and (3) arithmetic processing. Therefore our symbol processing model is augmented to allow such queries. Our model has been embedded in the Refine language [Refine-87]. This is a wide-spectrum very high level language, which provides arithmetic, set-theoretic operations, logic with universal and existential quantification, program transformation constructs, and object-oriented programming with single inheritance. Note however, that all explicit relations in our model are pure entity-relational. This added query power is convenient when defining the implicit LOOKUP relation.

Our entity-relational symbol processing model is currently implemented as a set of mappings (associations) between objects. Since mappings can be defined between an object and a set of objects, or between a set of objects and another set of objects, many-to-many or one-to-many relations can readily be represented as mappings.

ENTITIES

```
Declobject:    subtype-of  Programobject
Region:        subtype-of  Programobject
Declaration:   subtype-of  Programobject
Statement:     subtype-of  Programobject
```

*Figure 5. Entities in the entity-relational model.*

BINARY RELATIONS/ASSOCIATIONS

Each binary relation is represented as a mapping together with its converse mapping. Since many-to-one mappings are not invertible, we can instead form a converse mapping, where the range consists of sets of entities. The REGION_OBJECT relation is sparse - not all declobjects have regions, and vice versa.

```
Relation  DEPENDENCY:
  USED_BY   : map  Declobject     -> set(Programobject)
  USES      : map  Programobject -> set(Declobject)
Comments
  USED_BY  The set of program objects which depend on a declobject.
  USES     The set of declobjects a program object is dependent on.
```

```
Relation  REGION_NESTING:
  PARENT_REGION  : map  Region  ->  Region
  CHILD_REGIONS  : map  Region  ->  set(Region)
Comments
  PARENT_REGION  The innermost surrounding region.
  CHILD_REGIONS  The set of child regions of a region.

Relation  DEFINED_WITHIN:
  DECLS_IN_REGION     :  map  Region      ->  seq(Declobject)
  REGION_AROUND_DECL  :  map  Declobject  ->  Region;
Comments
  DECLS_IN_REGION     The sequence of declobjects defined in a region.
  REGION_AROUND_DECL  The region within which a declobject is defined.

Relation  INHERIT:
  INHERITS       : map  Region  ->  Region
  INHERITED_BY   : map  Region  ->  set(Region)
Comments
  INHERITS       The region which exports declobjects into this region.
  INHERITED_BY   The set of regions by whom this region is inherited.

Relation  PRINT_NAME
  PRNAME         : map  Declobject  ->  Symbol
  PRNAME_OF      : map  Symbol      ->  set(Declobject)
Comments
  PRNAME         The printname of an object.
  PRNAME_OF      The declobject of which this is a printname.

Relation  REGION_OBJECT
  OBJ_REGION     : map  Declobject  ->  Region
  OBJ_REGION_OF  : map  Region  ->  Declobject
Comments
  OBJ_REGION       A region which may be associated with a declobject.
  OBJ_REGION_OF    The programobject which is associated with a region.
```

*Figure 6. Relations/associations in the entity-relational model.*

## BOOLEAN PROPERTIES OF REGIONS

```
TRANSPARENT
  Declobjects from within a transparent region are directly visible outside.
QUALIFIED
  Declobjects from within a qualified region are visible by qualification.
PARTIALLY_QUALIFIED
  Declobjects which are explicitly named as exported from within a region.
ORDINAL_QUALIFIED
  Declobject are visible by their ordinal position - procedure parameters.
PARTIALLY_NESTED
```

```
A partially nested region inside an outer region may introduce new
declobjects, but their names may not collide.
```

*Figure 7. Boolean properties of regions.*

PROPERTIES OF DECLOBJECTS

```
POSITION     : ProgramPoint
  Position of a declaration or program object. It can be represented as
  line number, character position, or tree node address.
TYPE_OF      : Typespec
  Type of a declaration object.
```

*Figure 8. Properties of declobjects and certain programobjects.*

We currently identify five boolean properties of regions: transparent, qualified, partially_qualified, ordinal_qualified and partially_nested regions. An example of a transparent region is the definition of the enumeration type Color, where all components are externally visible. A record declaration is a typical example of a qualified region - components defined within the region are externally visible by qualified access. A procedure declaration which introduces a block of local declarations is an example of a partially_nested region. Finally, a region which inherits another region, can extend the visibility of declarations from this other region to itself. The Pascal WITH-statement is an example of a region which inherits other regions. Thus, it makes the fields from some record declaration visible within itself.

For Pascal or Modula-2, type analysis is not needed for lookup or definition of symbols. However, for the Ada language, type information is necessary in order to resolve overloaded symbols.

The DEPENDENCY Relation:

```
Dependenton:    Used-by:
  <GotoXY,       {BoxProg,DrawBox}>
  <ClrScreen,    {BoxProg}>

  <BoxVar,       {BoxProg}>
  <DrawBox,      {BoxProg}>
  <BoxType,      {BoxVar}>

  <X1,           {BoxProg}>
  <Y1,           {BoxProg}>
  <X2,           {BoxProg}>
  <Y2,           {BoxProg}>

  <Drawline,     {DrawBox}>
  <X1,           {DrawBox}>
  <Y1,           {DrawBox}>
  <X2,           {DrawBox}>
  <Y2,           {DrawBox}>
  <Dx,           {DrawBox}>
  <Dy,           {DrawBox}>
```

```
The REGION_NESTING Relation:
```

```
parent_region:  child_regions:

  <R-Systemblock, {R-BoxProg}>
  <R-BoxProg,     {R-BoxType, R-Color, R-DrawBox, R-Boxvar_WITH}>
  <R-DrawBox,     {R-Drawline}>


The DEFINED_WITHIN Relation:
  parent_region:  decl_objects:

  <R-Systemblock  {GotoXY, ClrScreen}>
  <R-BoxProg,     {Boxtype, Boxvar, DrawBox}>
  <R-BoxType,     {X1, Y1, X2, Y2}>
  <R-Color,       {Red, Green, Blue}>
  <R-DrawBox,     {X1-2, Y1-2, X2-2, Y2-2, Dx, Dy, Drawline}>
  <R-Drawline,    {DeltaX, DeltaY}>

The PRINT_NAME Relation:

  Namestring:   Object:

  <"BoxProg",   BoxProg>
  <"BoxType",   BoxType>
  <"DrawBox",   DrawBox>
  <"X1",        X1>
  <"X1",        X1-2>
  <"Y1",        Y1>
  .....         .....


The INHERIT relation:
  Donor:        Inheritor:

  <Boxtype,     Boxvar-WITH>;
```

*Figure 9. An example of relations. The simplified version of the entity-relational model has been applied to the BoxProg program example. Note that when two objects have the same name, e.g. the record field X1 and the formal parameter X1, these objects are denoted X1 and X1-2 respectively.*

## 13. INFORMAL DESCRIPTION OF OPERATIONS

This section provides an informal description of the incremental symbol processing operations with emphasis on the lookup operation, which later in this paper is expressed formally.

A printname and a current region are input to most variants of the lookup operations. In addition, the current position need to be supplied for certain languages, and a typespec for overload resolution. The lookup will return a matching declaration object, or a special *Undefined* value.

*Qualified lookup*

A typical example of qualified lookup is the lookup of record field names which are qualified by a record variable using dot-notation, e.g. Recordvariable.Fieldname, or the lookup of procedures defined in some Modula-2 module, e.g. Modulename.Procname. It is conceptually simple: perform the lookup among declaration objects defined within the record region or the module region. This region is here denoted by the current region.

In relational terminology this can be expressed using our relations PRINT_NAME and DEFINED_WITHIN. Find all declaration objects obj, such that DEFINED_WITHIN(obj, current_region) and PRINT_NAME(printname, obj). This can be conceptually performed by first joining the relations PRINT_NAME and DEFINED_WITHIN, and then performing a selection using printname and current_region as a combined key. The search can be made more efficient by performing most of the selection operations before the join, or maintaining a precomputed joined relation.

*Visibility lookup*

This is a lookup operation where the given name directly implies some declaration object which is visible in the current context. The visibility of declaration objects is the primary criterium in this lookup. Typical examples are the lookup of a variable name, a procedure name, or a type name.

However, several factors complicate the lookup operation. Declarative regions are usually nested, and declarations in inner regions may redefine symbols with the same name in outer regions. Certain regions inherit declarations from other regions. For example, the Pascal WITH-statement inherits field declarations from some record type region. Certain regions are transparent, e.g. Pascal enumeration type declarations, which makes their components visible in the parent region.

A simple, but perhaps inefficient, way of performing the lookup is as follows. Use the current printname to index the PRINT_NAME relation in order to select all objects with that printname. Then use the DEFINED_WITHIN relation to find within which region each such object is defined. Now the second phase of the lookup starts. We must find the set of regions which have declarations that can be visible from the current region. This includes the parent region, the parent of the parent region etc., following the REGION_NESTING relation to the uppermost region. It also includes transparent child regions, and regions which are inherited by such regions.

First example:

Here the current printname is "X1" and the current region is DrawBox. Selections using "X1" as an index into PRINT_NAME, yield two objects: X1 and X1-2. The DEFINED_WITHIN relation further yields X1-2 which is defined within the DrawBox region and X1 which is defined within the Boxtype region. Then we look for the set of possibly relevant regions in addition to DrawBox. Using REGION_NESTING, we obtain the parent region BoxProg and Systemblock. The child region Color is transparent and should also be included. Thus, the set of possible regions becomes {`DrawBox`, `BoxProg`, `Systemblock`, `Color`}. Finally we intersect with the regions of our two possible objects, eliminating X1 since it is defined within Boxtype. Thus the final result of our lookup is X1-2.

Second example:

The current printname is again "X1", but we now perform the lookup from a position in the WITH-Boxvar statement region, which thus is the current region. As in the previous example, "X1" first matches the two objects X1 and X1-2. Using the nesting hierarchy we then obtain {`WITH-Boxvar`, `BoxProg`, `SystemBlock`} as possible defining regions. The Boxtype

region is added since it is inherited by the WITH-Boxvar region - see the INHERIT relation. The Color region is also added since it is transparent and is a child of BoxProg. This yields the set {`WITH-Boxvar, BoxProg, SystemBlock, Boxtype, Color`}. Since X1 is defined within Boxtype, but X1-2 is defined within Drawline, the final result is X1.

So far we have ignored the possibility that declarations in inner blocks redefine symbols which are declared in outer blocks. Such ambiguities can be resolved by introducing an auxiliary relation REDEFINED between declaration objects [Reiss-83]. However, the random access nature of incremental symbol processing makes REDEFINED somewhat unsuitable, since the contents of this relation is dependent of the current focus of processing. In the worst case REDEFINED would have to be rebuilt at each lookup. A more efficient solution is used in the precise formulation of the lookup algorihm further on in this paper.

*Define symbol*

This operation associates a symbol in the current context with a new declaration object. First, create a declaration object and elaborate type information specified by the declaration which defines the symbol. This means updating the PRINT_NAME relation with a tuple `<printname, object>` and some other object attributes. Then do a partial lookup of the symbol in the current region, and in regions which are transparent to or inherited into the current region. If the lookup succeeds, then there are multiple definitions of the symbol, in which case the new definition is illegal.

Otherwise, enter the symbol and its declaration object into the current region in the symbol table. This means inserting a tuple `<object, current_region>` in the relation DEFINED_WITHIN. If the new declaration introduces a region, e.g. procedure, function, record- declarations, the object also belongs to the class of region objects. Then the relations REGION_NESTING and REGION_OBJECT should be updated. The INHERIT relation should be updated if the new region inherits declarations from some other region. If the new region object contains component declaration, then we first have to introduce this new region. For example, the symcursor denoting the current region should be set to this new region before the components are defined.

*Navigation operations*

*Enter region* will set the current region to be the region which is to be entered. Subsequent definition operations will insert definitions within the current region. Lookup operations start searching for object definitions within the current region. *Leave region* is really a special case of Enter region, since it means that we should enter the parent region of the current region. The parent region is found by indexing the REGION_NESTING relation using the current region object as a key.

*Declaration update*

As mentioned before, the insert operation is essentially a define operation. However, in an incremental symbol table we have to check if the inserted declaration will hide some existing declaration which is already used in the current context. This is done by performing a full lookup on the symbol at the definition point - not a partial lookup as in an ordinary define operation. Suppose the new definition partially hides an existing declaration which is

referenced in the current context. In such a case, all objects in the current context which are dependent on the previous declaration have to be re-elaborated and recompiled. The checking is performed by doing a lookup.

To delete a declaration, remove the symbol and its associated declaration object from the symbol table, i.e. update the relations PRINT_NAME and DEFINED_WITHIN. If the removed object has a region, then also update REGION_NESTING and INHERIT. Use the removed object as a key to the DEPENDENCY relation to find all dependent objects, and mark those for incremental recompilation. Also mark transitively dependent objects in the same way.

*Dependency update*

For each new use of a declaration object, update the dependency structure to reflect this use by inserting a tuple `<current_region, object>` into the DEPENDENCY relation. If a use of a declaration object is removed, and there are no more uses of that object within the current region, then delete the tuple `<current_region, object>` from the DEPENDENCY relation.

*Dependency Query*

This query answers the question: *Who depends on me?* Return the set of all declaration objects which are dependent on a certain object by making a selection from the DEPENDENCY relation, using this object as a key. Many other query variants can be expressed using the relational algebra.

## 14. AN EXTENDED ENTITY-RELATIONAL MODEL

Our simple incremental relational symbol processing model cannot exactly represent languages with single pass symbol processing semantics, where a symbol cannot be referenced before it has been defined. This problem can be solved by introducing the notion of *Position* for declaration objects.

Another problem is that formal parameters can be accessed outside their defining procedures by qualification on their ordinal positions in parameter lists. In addition to ordinal qualification, languages such as Ada also support named qualification on parameter names. Thus, we need to extend our previous model, which for each procedure has a single declarative region for both parameters and local variables. In the extended model we introduce at least two declarative regions for each procedure: one qualified region enclosing the whole procedure including the formal parameters, and one or more additional regions nested within, for blocks with local declarations.

This model suits Ada and C, where declarations in nested local blocks are allowed to hide parameter declarations. In Pascal however, local declarations are not allowed to hide parameter declarations - they are conceptually part of the same block. Our solution to this problem is to introduce another class of regions called *partially nested regions*. This kind of region is similar to nested regions except that declarations within partially nested regions are not allowed to hide declarations from outer regions.

Remember that the *scope* of a declaration object consists of the parts of a program where it is legal to reference the declaration. It can be expressed as a function of simpler notions such as position, nesting of regions, visibility rules etc. This is expressible within our extended model. Scope need not conform to the nice nesting structure of programming languages. For example, in the C program example in Appendix A, the scope XX1 of the global variable XX covers the initial part of the body of the function foofunc, whereas the scope XX2 of the local variable XX covers the rest of foofunc.

## 15. PRECISE FORMULATION OF THE LOOKUP ALGORITHM

There are two basic variants of the lookup operation, which define the implicit LOOKUP relation. Those two are lookup by direct visibility and lookup by qualification. Visibility lookup is the most common case, where an occurrence of a name in the current context should be associated with some declared program entity. This can be a complex function of nesting structure, local context, positions of declarations, import and export of declarations, and types when there is a possibility of overloading. Simple overload resolution with type comparison is included in the present lookup algorithm. A complete overload resolution algorithm for Ada is presented by [Baker-82].

The function Visibility_Lookup starts searching within the innermost region that includes the current program point. If there is no match, continue at the next outer level in the region nesting hierarchy. According to the nesting rules only the first match should be returned. Thus positional and type-checking predicates, when relevant, must be applied during the search. The Undefined value is returned when there is no match. Note that at each nesting level we consider both declarations from the current region, declarations from regions which are inherited into the current region (e.g. Pascal WITH), and declarations from transparent child-regions (e.g. Pascal Enumeration). PRNAME_OF(Name) returns the set of declaration objects with a certain print name. For single-pass compiled languages such as Pascal and C, which require that a definition occurs before its use, the Check_position predicate is relevant.

Lookup by qualification is far simpler. Just lookup the qualified name within the region where it is defined. For example, in the common case of a record field reference, search for it within the declarative region of the record type. A special case is ordinal qualification, where instead of a name, an integer index denotes the declaration object. For certain languages [Refine-87] it is possible to denote a fieldname in this way, e.g. xrec.2 would reference the second field in xrec. However, ordinal lookup is most commonly used for accessing procedure parameters to make type checking possible at call sites.

```
FUNCTION  Visibility_Lookup(Name           : Symbol;
                            Currentregion  : Region;
                            Currentposition: ProgramPoint;
                            Currenttype    : Typespec): Declobject
(* Lookup of Name in the current context, which is denoted by Currentregion.
   Start Looking in the current region, and all regions (transparent or
   inherited) which are visible at the current nesting level.
   Continue searching outwards at each level in the nesting hierarchy,
   until a matching declaration object is found, or
```

```
  the root, which has no parent, is encountered.
*)
BEGIN
     r  :=  Currentregion;
   REPEAT
        rset  :=  {r} union {INHERITS(r)} union
                          {x | x in CHILD_REGIONS(r) and TRANSPARENT(x) }
     FOR  d:Declobject  IN  PRNAME_OF(Name)  DO
        IF  REGION_AROUND_DECL(d) in  rset
             and  Check_Type(d, Currenttype)
             and  Check_Position(d, Currentposition)
        THEN
            RETURN d;
     END FOR;

     r  :=  PARENT_REGION(r);
   UNTIL  Undefined(r);

   RETURN  Undefined;
END;


FUNCTION  Check_Position(d: Declobject; p: Position): Boolean;
(* Predicate to check if d is accessible before or at position p.
   For languages which require define before use *)
BEGIN
  RETURN  BEFORE(POSITION(d), p);
END;


FUNCTION  Check_Type(d: Declobject; t: Typespec): Boolean;
(* Predicate to check if d is compatible with typespec t.
   For languages which allow overloading with respect to type/signature *)
BEGIN
  RETURN  TypeCompatible(TYPEOF(d), t);
END;



FUNCTION  Qualified_Lookup(Name            : Symbol;
                           Currentregion   : Region;
                           Currenttype     : Typespec): Declobject;
(* Qualified lookup of Name, example:  Qualifyname.Name
   Currentregion can be obtained by OBJ_REGION(Qualifyobject), where
   Qualifyobject is the declaration object associated with Qualifyname.
   Currenttype is useful when overloading is a possibility.
 *)
BEGIN
   FOR  d:Declobject  in  DECLS_IN_REGION(Qualifyregion)  DO
      IF  PRNAME(d) = Name;
          and  Check_Type(d, Currenttype)
      THEN
          RETURN d;
   END FOR;
   RETURN Undefined;
END;



FUNCTION    Ordinal_Lookup(Ordernumber   : Integer;
                           Qualifyregion : Region): Declobject;
(* Lookup by ordinal qualification. Example:  lookup of a procedure
   parameter for type checking at some call site.  In this case
   Qualifyregion can be obtained by REGION-OF(Procedureobject).
 *)
BEGIN
   RETURN  the element extracted by
           indexing the Ordernumber:th element in the sequence
           DECLS_IN_REGION(Qualifyregion);
END;
```

*Figure 10.*

The lookup procedures for our incremental symbol processing model are shown in pseudo code in Figure 10. Note that the procedures are fairly short in this model, despite the apparent complexity of lookup. The Refine implementation of this pseudo code has approximately the same length.

## 16. WHY IS THIS ENTITY-RELATIONAL MODEL INCREMENTAL?

Incremental means that the work needed to introduce a change is more or less proportional to the size of the change. Thus it is essential that the datastructures of the model, in our case relations, *permit efficient updating* - both insertions and deletions. Also, the *dependency* relation permits the effects of changes to be propagated only to affected program entities, without affecting others. *Direct access* and order-independent processing: changes can be processed in any order anywhere in the program. The symbol table is *independent of the current focus of processing.* For example, if we introduced a relation REDEFINED - that represents which declarations have been redefined in the current nesting context - it would violate this rule, since it would need to change if the current focus changed.

## 17. REPRESENTATION OF CERTAIN LANGUAGE CONSTRUCTS

In this section we show how some special language constructs can be represented in the entity-relational symbol processing model. The current list of examples are from Pascal, Modula-2 and Ada. Regions are marked in these examples.

Pascal declaration of enumeration scalars

This construct introduces a set of named entities within a small region of the program. This is easily modelled by a TRANSPARENT region, which causes entities from inside it to be directly visible in the surrounding region.

```
Example:  TYPE Color = (Red, Blue, Green);
                       ---------------
```

Pascal record type declaration

A record type declaration introduces a number of field entities. If the body of the declaration is modelled by a QUALIFIED REGION, then the fields will be visible by qualification in the surrounding region.

```
Example:  TYPE Personrec = RECORD  Age:Integer; Weight:Integer  END;
                           ---------------------------------------
```

Pascal WITH statement

A Pascal WITH statement causes the fields of a record variable to be directly visible within its body. This is modelled by a region around its body. This regions INHERITS the region around the record type where the fields are declared. Thus the fields become directly visible.

Two nested regions are associated with this example WITH-statement. The xrec record type region is imported/inherited by the outer WITH-region, and the yrec region by the inner region.

```
Example:   WITH xrec,yrec DO .... ;
                        -------------- Outer region
                             --------- Inner region
```

Modula-2 IMPORT declaration, Ada WITH-declaration

Even though it sounds strange, we introduce a small QUALIFIED REGION for each module identifier. Each such region covers only the identifier itself, and INHERITS the region around the module/package specification part. Thus all declaration objects from within the module/package are available within this small region. Since it is a QUALIFIED REGION, declarations are also visible outside it by qualified access.

In this example there are two small qualified regions, where each only covers the relevant module/package identifier. Xmodule is inherited by the first region, and ymodule by the second.

```
Examples:
  Modula-2:   IMPORT xmodule,ymodule;
  Ada      :  WITH   xmodule,ymodule;
                     ------- -------
                    region 1  region 2
```

Modula-2: Selective import statement

Here, a PARTIALLY TRANSPARENT region is introduced around the list of identifiers. The xmodule region is inherited by this underlined partially transparent region. Thus only these explicitly mentioned identifiers are made directly visible outside this region.

```
Example:
  Modula-2:   FROM xmodule IMPORT  foo1,foo2,foo3;
                                   --------------
```

Ada USE declaration

This case is similar to the Ada WITH-declaration, but here a TRANSPARENT region is introduced around each identifier. Thus declarations from inherited regions are made directly visible.

In our example we introduce a sequence of two transparent regions, where the first inherits xmodule and the second inherits ymodule. All definitions from these modules are made directly visible outside the transparent regions.

```
Example:
  Ada   :      USE xmodule, ymodule;
                   -------  -------
```

## 18. CORRECTNESS

In this section we discuss correctness of queries and very high level specifications, where queries on our symbol processing model is one special case. Since the transformations from specification to executable code are automatic, correctness of executable programs follows automatically if the transformations have been proved correct, and if the specification is correct with respect to intuitions. In the following, we will use the normal definition of correctness which only requires that the implementation has the same semantics as the specification, without taking intuitions into account.

Data type refinement is an especially important program transformation from our point of view, since it allows transformation of relational queries to query operations on other data structures which may be more efficient in certain respects. A special case of such data structure refinement is described in [Horwitz,Teitelbaum-86] where queries on implicit relations are transformed into queries on tree structures. In the rest of this section we will focus on how to define correctness of data type refinements, which follows the treatment in [Goldberg,Kotik-83].

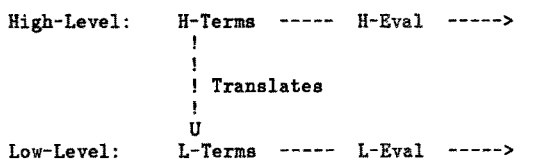


Figure 11. Correctness of a data-type refinement, expressed through a commutating diagram.
A refinement is a pair I = <Translates, Abstracts>. It is correct iff: forall t1 in H-Terms, forall t2 in L-Terms
[(t1 --> t2) => (H-Eval(t1) = Abstracts(L-Eval(t2))) ]

Informally, we can define a data type D as a pair $\texttt{<Operations}_{D}, \texttt{ DataValues}_{D}\texttt{>}$. We also assume the existence of a function *Eval*, which maps terms into terms, where a term is an expression involving only the operations of some data type. A *refinement* I of a type H = $\texttt{<Operations}_{H}, \texttt{ DataValues}_{H}\texttt{>}$ to a type L = $\texttt{<Operations}_{L}, \texttt{ DataValues}_{L}\texttt{>}$, is a pair <Translates, Abstracts>. *Translates* is a relation between H-Terms and L-Terms. It specifies the possible translations from expressions in H-Terms to expressions in L-Terms. *Abstracts* is an abstraction map, that maps each value in $\texttt{DataValues}_{L}$ to the unique value in $\texttt{DataValues}_{H}$, which it represents in the refinement, and is the identity function for all other values. Remember that there are usually several lower level concrete representations for each higher level abstract data type.

It is then natural to define a data type refinement as correct, if translated expressions always preserve the semantics of the original expressions. This is the same as requiring that the diagram in Figure 11 commutes. More formally, a refinement I = <Translates, Abstracts> is correct if

```
forall t1 in H-Terms, forall t2 in L-Terms
[(t1 --> t2) => (H-Eval(t1) = Abstracts(L-Eval(t2))) ]
```

Refinements can be composed under certain conditions. By the above definition it is easy to realize that the composition of two correct refinements is itself correct.

## 19. COMPARISON WITH PREVIOUS WORK

The DICE hierarchical incremental symbol processing model [Fritzson-85], is too specialized, and lacks a general declarative query language. A more general incremental model is clearly needed.

The [Reiss-83] paper presents a general model for the generation of non-incremental symbol processing mechanisms from ad-hoc declarative specifications. In addition, that paper contains a separate formal relational symbol processing model. However, that relational model is not suitable as a basis for generation of efficient symbol processing mechanisms. Also, that relational model is not incremental, and it does not correctly model the fact that for many languages the scope of a declaration extends forward from the actual point of declaration. However, its non-relational implementation still works correctly because of the sequential processing nature of non-incremental symbol processing.

The present entity-relational model is an improvement in several respects. It is incremental and it can be used to generate efficient symbol processing mechanisms from high-level declarative specifications through transformations. It is also conceptually simplified, e.g. the complex notion of scope group [Reiss-83] is eliminated. Our model uses the more precise notion of declarative region as a basis for expressing scope. We also include the notion of position, which is needed in an incremental context.

The PSG system [Bahlke,Snelting-86], has the possibility of generating simple scope analysis. However, PSG context relations are primarily designed for use on type analysis, which includes the reconstruction of types from unification on incomplete program fragments.

Attributed-relational grammars [Horwitz,Teitelbaum-86] appears useful as a possible means of communication between a language-based editor and our entity-relational model. However, only very simple scope analysis for a Pascal subset is mentioned in that paper. It could clearly be extended by integrating the entity-relational model presented in this paper.

## 20. FUTURE WORK

We are currently planning to use similar transformational techniques to generate other parts of compilers than the symbol processing module.

Finite differencing techniques [Paige,Koenig-82] have so far been used to transform powerful set-theoretic operations to cheap incremental counterparts. We are considering the investigation of finite differencing techniques in order to compile code that will cheaply and incrementally update relational expressions and maintain invariants, after small updates to basic input relations.

Another interesting area concerns the extension of our current incremental model to support programming-in-the-large: version handling and configuration control. A design for a distributed network version of our relational program database is desirable. Solve the efficiency problems of current relational databases, for example by relaxing the consistency requirement at the single tuple level, and use invisible caching techniques. There is the question if clustering, caching, query-optimization and relaxed consistency requirements are enough to achieve good performance?

## APPENDIX A - C LANGUAGE EXAMPLE OF SCOPE, REGION, POSITION

The scope of a declaration object consists of the parts of a program where it is legal to reference the declaration. Scope need not conform to the nice nesting structure of programming languages. For example, in the C program example below the scope XX1 of the global variable XX covers the initial part of the body of the function foofunc, whereas the scope XX2 of the local variable XX covers the rest of foofunc.

The example below contains six declarative regions: R-File for the whole file, R-foofunc for the function foofunc, and R-block for the body of foofunc. R-foofunc is a qualified region to provide positional access to parameters from the rest of the program; R-block is a nested region for the body of the function.

```
R-File region
 !
 !
 !              int   XX = 3;                -!
 !                                            !  Scope XX1 for XX (int)
 !  R-foofunc   void foofunc(ch)              !
 !    !           char ch;                    !
 !    ! R-block {                             !
 !    !   !       int     ZZ = XX;           -!                    <-- Position P1
 !    !   !
 !    !   !       double  XX;                -!                    <-- Position P2
 !    !   !                                    !  Scope XX2 for XX (double)
 !    !   !       ....                         !
 !    !   !     }                             -!
 !
 !
 !
 !   R-X1:       define   X  3                -!
 !    !            ....                        !  Scope X1 for X
 !    !            ....                       -!
 !
 !   R-X2:       undef    X                   -!
 !    !            ....                        !  Scope X2 for X
 !    !            ....                       -!
 !
 !   R-X3:       define   X  10               -!
 !    !            ....                        !  Scope X3 for X
 !    !
```

*Figure 12: C language program example*

```
The DEPENDENCY Relation:

  Dependenton:    Used-by:
  <XX,            {foofunc}>


The REGION_NESTING Relation:

  parent_region:  child_regions:

  <R-Systemblock,    {R-File}>
  <R-File,           {R-foofunc}>
  <R-foofunc,        {R-block}>
  <R-file,           {R-X1, R-X2, R-X3}>


The DEFINED_WITHIN Relation:
  parent_region:  decl_objects:
```

```
  <R-File,          {XX, foofunc}>
  <R-foofunc,       {ch}>
  <R-block,         {ZZ, XX-2}>       ;  the second XX declaration
  <R-File,          {X, X-2, X-3}>    ;  the second X = undef
                                      ;  the third  X = 10

The PRINT_NAME Relation:

  printname:   object:

  <"XX"         XX>
  <"XX"         XX-2>
  <"ch"         ch>
  <"ZZ"         ZZ>
  <"foofunc"    foofunc>
  <"X"          X>
  <"X"          X-2>
  <"X"          X-3>


The INHERIT relation:
  Donor:        Inheritor:

   ....empty....
```

*Figure 13. Relational symbol table for the C language program of figure 12.*

REFERENCES

[ADA-83] United States Department of Defense, "Reference Manual for the Ada Programming Language", ANSI/MIL-STD-1815A/1983, February 17, 1983.

[Bahlke,Snelting-86] Rolf Bahlke, Gregor Snelting: "The PSG System: From Formal Language Definitions to Interactive Programming Environments", TOPLAS Vol 8, No 4, October 1986.

[Baker-82] T. P. Baker: "A One-Pass Algorithm for Overload Resolution in Ada", ACM TRansactions on Programming Languages and Systems, Vol. 4, No 4, Oct. 1982, pp 601-614.

[Birtwistle,et.al-73] G M Birtwistle, O-J Dahl, B Myhrhaug, K Nygaard: "SIMULA BEGIN", 391 pp, AUERBACH Publishers Inc, Philadelpia, Pa., 1973.

[Chen-76] Peter Pin-Shan Chen· "The Entity-Relationship Model - Towards a Unified View of Data", ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976.

[Conradi,Wanvik-85] Reidar Conradi and Dag Wanvik, "Mechanisms and Tools for Separate Compilation", Technical Report No 25/85, Nov 1985, The University of Trondheim, Division of Computer Science, N-7034 Trondheim-NTH.

[Fritzson-83] Peter Fritzson, "Symbolic Debugging Through Incremental Compilation in an Integrated Environment", The Journal of Systems and Software 3, 285-294 (1983).

[Fritzson-85] "The Architecture of an Incremental Programming Environment and some Notions of Consistency" Workshop on Software Engineering Environments for Programming-in-the-Large, Harwichport, Massachusetts, June 9-12, 1985.

[Goldberg,Kotik-83] Allen Goldberg and Gordon Kotik: "Knowledge-Based Programming: An Overview of Data Structure Selection and Control Structure Refinement", KES.U.83.7, November 1983, Kestrel Institute, 1801 Page Mill Road, Palo Alto, CA 94304. Also in: Software Validation, H.L. Hausen, Ed., North-Holland, 1984.

[Harbison,Steele-84] Samuel P. Harbison and Guy L. Steele Jr., "C - A Reference Manual", Prentice-Hall, 1984.

[Hoover-86] Roger Hoover: "Dynamically Bypassing Copy Rule Chains in Attribute Grammars", 13:th Annual ACM Symposium on the Principles of Programming Languages, St. Petersburg, Florida, Jan 1986.

[Horwitz,Teitelbaum] Susan Horwitz, Tim Teitelbaum: "Generating Editing Environments Based on Relations and Attributes", ACM Transactions on Programming Languages and Systems, Vol. 8, No. 4, October 1986.

[Linton-84] M. A. Linton: "Implementing Relational Views of Programs" In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (Pittsburgh, Pa., April 1984), ACM New York, 132-140.

[Paige,Koenig-82] Robert Paige, Shaye Koenig: "Finite Differencing of Computable Expressions", TOPLAS 4.3, July 1982, pp 402-454.

[Rational-86] "Private communication on the Rational Incremental Ada Compiler" Rational, 1501 Salado Drive, Mountain View, California 94043.

[Rational-85] James E. Archer, Michael T. Devlin: "Rationals Experience Using Ada for Very Large Systems", Proc of the First International Conference on Ada Programming Language Applications for the NASA Space Station, Houston, Texas, June 2-5, 1986.

[Refine-87] Reasoning Systems: "Refine$^{TM}$ User's Guide", Version 2.0, September 1987. Reasoning Systems Inc., 1801 Page Mill Rd., Palo Alto, CA 94304.

[Reiss-83] Steven P. Reiss, "Generation of Compiler Symbol Processing Mechanisms from Specifications", ACM TOPLAS 5.2 April 1983.

[Reps-83] Thomas W. Reps: "Generating Language-Based Environments", The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusets 02142

[Smith,et.al] Douglas Smith, Gordon Kotik, Stephen Westfold: "Research on Knowledge-Based Software Environments at Kestrel Institute", IEEE Trans. on Software Engineering, Vol SE-11, No 11, Nov 1985.

[Uhl,et.al-82] J. Uhl, S. Drossopoulou, G. Persch, G. Goos, M. Dausmann, G. Winterstein, W. Kirchgässner: "An Attribute Grammar for the Semantic Analysis of Ada", IX, 511 pages, Lecture Notes in Computer Science, Springer Verlag, 1982.

[Zdonik,Wegner-85] Stanley B. Zdonik and Peter Wegner, "A Database Approach to Languages, Libraries and Environments", Workshop on Software Engineering Environments for Programming-in-the-Large, Harwichport, Massachusetts, June 9-12, 1985.