Smart Recompilation in the XE Compiler
Extended Abstract

Esko Nuutila
Vesa Hirvisalo
Jari Arkko
Juha Kuusela
Markku Tamminen

Helsinki University of Technology
Laboratory of Information Processing Science
02150 Espoo, Finland

Abstract

This article describes the smart recompilation in a compiler for the XE programming language. XE is a data abstraction language that resembles Clu. The problem of smart recompilation consists of the detection of the program units that must be recompiled after a change to some program unit and the detection of compilation order for those program units. The recompilation algorithms of the XE compiler are described and compared with other recompilation algorithms.

1 Introduction

The XE language and its programming environment have been developed within the ExBed project, which aims to support the construction of embedded expert systems. In the development of the XE compiler one of the major goals was to perform as little recompilation as possible in response to changes in source modules, without any user assistance. We describe a simple solution to this problem and compare it to other proposals. Although the algorithm was designed for the recompilation of XE programs it can be used in the recompilation of other languages. We also describe shortly the XE language and its compiler.

The structure of the article is following. In section 2 we describe shortly the XE language and the compiler. Also the problem of compilation order is discussed. In section 3 we present the XE recompilation algorithm and compare it with other known recompilation algorithms.

2 Background

2.1 The XE Language

XE is a data abstraction language descending from Clu [Liskov 86], without being its direct extension. An XE program is made up of one or more program units. Each program unit is a *procedure*, *iterator*, *rule*, *data type*, or *constant*.

Program units can have a nested structure relative to one another. For example, it is possible to nest a procedure definition inside another one to obtain an internal procedure. XE has no global variables, but variables and arguments of the surrounding unit can be bound by the nested unit. Thus it is possible to write lexical closures in XE.

All program units, except constants, can be parameterized. This allows a set of related abstractions to be defined by a single unit. A parameterized program unit is called a *generator*, because it defines an ordinary program unit each time it is provided with legal actual parameters; for example `array[int]` and `array[string]`. XE program units can also be parameterized by *field lists*, i.e., lists of name-type pairs, such as `record{name: string, age: int}`.

Each program unit has a *header* that specifies the interface (signature) of the unit and a *body* containing the implementation. The implementation is not visible to other program units. As in Clu, no special interface or specification units are used.

2.2 The XE Compiler

The XE environment supports separate compilation of program units. Nested subunits, e.g., routines of a data type or local procedures are also compiled separately. The compilation of a program unit is not independent of the compilation of other units because program units refer to each other and the legality of these references must be checked. Therefore the compile-time information of the program unit must be stored so that it can be later accessed when compiling other units.

The compile-time information of a program unit is stored in a descriptor. The descriptor contains also the parse tree of the unit and compilation dependencies, i.e., a list of units that this unit refers to, and a list of units that refer to this unit. The descriptors are stored in a symbol table. The symbol table and the descriptors form a persistent compilation environment that remains in the virtual memory during a program development session and that is updated by each compilation.

The phase structure of the XE compiler is conventional. The front end consists of a scanner, a parser that builds an explicit parse tree, and a checker that verifies that context conditions are satisfied and updates the compilation environment. Currently the compiler has two back ends, one that produces Lisp code and another that produces Amsterdam Compiler Kit intermediate code EM.

2.3 Compilation Order

A set of program units cannot be compiled (or recompiled) in arbitrary order. The information that is needed in the compilation of a program unit must be computed before the unit is compiled. The compilation order can be determined from dependencies between program units. The usual rule for determining the order is that if program unit A depends on program unit B then B must be compiled before A. If the programming language permits recursive dependencies between modules this rule cannot always be used.

Hood, Kennedy and Müller describe the use of a compilation dependency graph in determining the order of recompilations. The nodes of the graph are modules and the arcs denote dependencies between modules. A general compilation dependency graph is cyclic, but it can be reduced to an acyclic graph where the nodes are the strongly connected components of the first graph. The acyclic graph is then ordered by topological sorting. The modules in a strongly connected component must be merged and compiled as a single unit ([Hood 86])

A similar method can be used when compiling a set of program units for the first time. In this case the program units must be preprocessed so that the compilation dependencies can be detected.

Various polymorphic features of XE make this method rather unsuitable for XE. Compilation dependencies cannot be easily detected before the actual compilation. Therefore we decided to use a quite different approach that is based on the idea of demand driven (or "lazy") computations. No compilation order is computed before compilation. Instead, the compiler starts to compile an arbitrary program unit and when a reference to another unit, that is not already compiled, is detected the compilation of the first unit is suspended and the compilation of the second unit is started. When that compilation has completed the compilation of the first unit is resumed. In order to avoid deadlocks in case of recursive dependencies, all parts of the second unit are not compiled. Only those parts that are needed in the compilation of the first unit are computed. It should be noted that no separate algorithm is needed for solving cyclic dependencies as in [Hood 86]. According to our experiences the average number of suspended compilations is small.

3 Smart Recompilation

The problem of maintaining consistency emerges when program units are compiled separately and semantic checks are performed across unit boundaries. When some program unit is changed that unit and all units that depend on it must be recompiled so that all changes have their desired effects.

The detection of the set of program units that must be compiled is tedious and therefore algorithms have been implemented for recompilation. The MAKE program [Feldman 79] is one of the simplest recompilation programs. A file is recompiled if it is changed or any file that it depends on is changed. The programmer has to enumerate all dependencies explicitly.

The problem with MAKE is that changes are not analysed in order to see what program units within a file have changed and which other units see the changes. Also the dependence relation is too coarse and causes unnecessary recompilations. For example, adding a new comment to some program unit in a file causes the recompilation of that file and all files that contain program units that depend on some units of this file.

3.1 Smart recompilation in the XE compiler

The smart recompilation algorithm in the XE environment determines the set of program units that are affected when some program units are changed. The algorithm is phase oriented, which means that for all affected program units it is known what phases of the compiler need be run in recompilation. No compilation order need be explicitly determined for the affected program units because the demand driven compilation method, that was described in section 2.3, is used also in recompilations.

3.1.1 The algorithm

The algorithm is based on compilation dependencies that the checker and other phases of the compiler detect. A *header dependency* is established from program unit A to program unit B if B is referred to in A's header. In a similar manner a *body dependency* is established from A to B if B is referred to in A's body. The code generator (or actually a separate phase that does inline coding) detects *inline dependencies*. If the call of routine B is inline coded in A then an inline dependency is established from A to B. It should be noted that an inline dependency never exists alone, but is always combined with a body dependency.

The detection of the affected program units consists of two phases. In the first phase the edit changes are analysed and a *change set* is computed in a similar way as in Tichy's algorithm [Tichy 86]. In the second phase the changes are propagated using the compilation dependencies and the set of affected program units is computed.

The edit change analysis is given two sets of parse trees $T_{current}$ and T_{old}. They are, for example, the results of parsing the old contents of and an editor buffer and the current contents of an editor buffer. These sets are compared and three sets T_{new}, $T_{deleted}$, and $T_{modified}$ are computed. T_{new} contains the parse trees that are in $T_{current}$ but not in T_{old}. $T_{deleted}$ contains the parse trees that are in T_{old} but not in $T_{current}$. $T_{modified}$ contains the parse trees that are both in $T_{current}$ and in T_{old} but that are not equal. For parse trees that are in $T_{modified}$ the type of the change is also computed. It is *header* if the header was changed and *body* if only the body was changed.

Change propagation uses T_{new}, $T_{deleted}$, and $T_{modified}$ and computes three sets of program units P_{new}, $P_{deleted}$, and $P_{recompile}$. P_{new} contains new program units that should be compiled, one for each parse tree in T_{new}. $P_{deleted}$ contains program units that should be deleted from the symbol table. The program units whose parse trees are in $T_{deleted}$ are in $P_{deleted}$, as well as the subunits of those units. $P_{recompile}$ contains the program units that should be recompiled.

$P_{recompile}$ consists of three parts. P_{header} contains all program units that must be recompiled because the static semantics of their header may have changed. P_{body} contains all program units that must be recompiled because the static semantics of their body (but not the header) may have changed. P_{inline} contains all program units whose static semantics has not changed but for which code must be generated again because of inline dependencies. The algorithm cannot guarantee that these sets are minimal, i.e., that they contain only the program units that are really affected. The sets are computed using the following rules.

Rule1 If the parse tree of program unit P is in $T_{modified}$ and the change type is *header* then P is in P_{header}.

Rule2 If the parse tree of program unit P is in $T_{modified}$ and the change type is *body* then P is in P_{body}.

Rule3 If there is a header dependency from program unit P_1 to program unit P_2 and P_2 is either in P_{header} or in $P_{deleted}$ then P_1 is in P_{header}.

Rule4 If there is a body dependency from program unit P_1 to program unit P_2 and P_2 is either in P_{header} or in $P_{deleted}$ then P_1 is in P_{body}.

Rule5 If there is an inline dependency from program unit P_1 to program unit P_2 and P_2 is either in P_{body} or in P_{inline} then P_1 is in P_{inline}.

After change propagation the program units in P_{new} and $P_{recompile}$ are compiled. For units that are in P_{inline} the code generator is called. For other units the checker is first called and after that, if no errors were detected, the code generator is called. Only the body of the units in P_{body} has to be checked. The demand driven compilation method is used in the compilations. The user can be asked for confirmation before the actual recompilation. There is also a program that scans the units in $P_{recompile}$ and gives the user an opportunity to edit the units.

3.1.2 Analysis

Let us now analyse the correctness of the presented rules. We try to show that no affected program units are missing from P_{header}, P_{body}, and P_{inline}. We assume that the compiler has correctly collected all dependencies between the program units.

By definition the header is that part of a program unit which is visible to other program units and the body is that part which is not visible to other programs units. Visible here means having an effect on the static semantics of a referring program unit. The deletion of a program unit is therefore also visible to other units. The static semantics of the header of a program unit may have changed only if either the header of the unit is changed or the unit refers to another unit in its header and the static semantics of that unit have changed in a visible way. Rule 1 takes care of the former case and rule 3 of the latter case and therefore P_{header} is computed correctly. The static semantics of the body (but not the header) of a program unit may have changed only if either the body of the unit is changed or the unit refers to another unit in its body and the static semantics of that unit have changed in a visible way. Rule 2 takes care of the former case and rule 4 of the latter case, thus P_{body} is computed correctly. Code must be regenerated for a program unit whose static semantics have not changed only if the unit refers to another unit in its body and the reference was inline coded. Further, the static semantics of the referred unit must have changed in an invisible way or the semantics may have remained unchanged, but code has to be regenerated for that unit. Rule 5 takes care of this case, thus P_{inline} is computed correctly.

The algorithm filters out unnecessary recompilations quite well. There are only a few of cases in which the static semantic is not actually modified, but the recompilation algorithm believes that it is. These cases do not occur very often. Also it would probably be more expensive to detect these cases than to recompile.

3.2 Other Smart Recompilation Algorithms

[Dausmann 84] describes various recompilation schemes using different levels of change analysis. The most advanced scheme described considers dependencies between various attributes of entities in a compilation unit. A compilation unit can be, for example, a procedure, an entity can be a variable declaration, and an attribute can be the size of the variable. An attribute depends on another attribute if the value of the latter is used in the computation of the former. This dependency relation can be extended to a relation between compilation units, which is then used in the computation of the set of units that must be recompiled. If attributes are divided into classes according to the phase of the compiler that computes them and the compiler is partitioned carefully then complete recompilation is not always necessary, only affected phases need be run. No explicit algorithms are given and compilation order is not considered in the article. Our algorithm is phase oriented in the way that Dausmann suggests.

[Rain 84] describes the recompilation algorithm used in the implementation of the Mary2 language. The algorithm first recompiles the changed program unit. If the export interface of the unit changes, then each unit which explicitly imports the changed interface is recompiled. The same test is applied recursively to each of those recompilations. In practice, there will be one or two levels of recompilations in most cases ([Rain 84]). [Järvinen 88] shows that this method does not decrease recompilations very much in ADA because only dependencies that are introduced via WITH-clauses may be cut. The main difference between Rain's algorithm and ours is that Rain's algorithm does not separate the detection of affected program units from the actual recompilation. Also the grain size of recompilation is different.

[Tichy 86] presents a smart recompilation algorithm that solves a slightly different problem: Given a compilation unit M_0 (e.g., a program file) and a set of contexts (e.g., definition files) $M_1, \ldots, M_n$, assume that the configuration $\{M_0, \ldots, M_n\}$ is legal and was compiled successfully. If some context M_j is changed to $\overline{M}_j$, determine whether the new configuration $\{M_0, \ldots, \overline{M}_j, \ldots, M_n\}$ is legal and whether the configuration should be recompiled. The algorithm computes a change set for the changed context. The change set consists of three sets ADD, DEL, and MOD. ADD contains the identifiers declared in $\overline{M}_j$ but not in M_j, DEL the identifiers declared in M_j but not in $\overline{M}_j$, and MOD the identifiers declared in both M_j and $\overline{M}_j$, but whose declarations differ. The change set is then compared with transitively computed reference sets (the sets of identifiers declared in some context and transitively referenced in some other context or compilation unit) of the contexts to find out whether the new configuration is legal and whether recompilation is necessary. This procedure can be repeated for all relevant configurations. Our algorithm and Tichy's resemble each other in that both determine a change set by comparing the parse trees of the changed program units. Both algorithms also compute transitive closures of references (or dependencies), but the closures are used for different purposes. Tichy's algorithm is clearly oriented towards bigger compilation units than ours.

[Hood 86] presents a smart recompilation algorithm that determines both the set of compilation units that must be recompiled after a change to a given unit and the corresponding recompilation order. The algorithm uses the compilation dependency graph that we described in section 2.3. The nodes of the graph are visited in topological order to detect the affected modules, and dataflow analysis is used to determine whether a particular node should be visited. Also this algorithm computes a change set. The biggest difference to our algorithm is that the recompilation order is explicitly computed, in our algorithm that was not needed. The detection whether a change is visible to other units is easier in our algorithm because we divide the dependencies and changes into different classes. This can be done because our compilation units do not contain subordinate definitions that are visible to other units. Thus we can avoid the dataflow analysis of dependencies. Both algorithms can handle recursive dependencies.

4 Summary

The smart recompilation algorithm of the XE compiler has been presented and compared with other recompilation algorithms. The dependencies between program units are divided into classes (header dependencies, body dependencies and inline dependencies). Therefore the detection of the program units that are affected by a change to some program unit is very easy. Because of the demand driven compilation method, no compilation order needs to be explicitly computed for the affected program units. For these reasons the XE recompilation algorithm is simple compared to other recompilation algorithms.

The XE recompilation algorithm can adapted to other languages. The dependency classes may vary for different programming languages. For some languages also the compilation units must be redefined, so that the dependencies can be classified. For example if the method is used in a Modula 2 compiler, the compilation units must be procedures, types, etc. instead of Modula 2 modules.

Acknowledgements

The ExBed project has been funded by the Technology Development Centre together with three private companies.

References

[Dausmann 84] Dausmann, M., Reducing Recompilation Costs for Software Systems in Ada, in System Implementation Languages: Experience and Assessment, Proceedings of the IFIP WG2.4 Conference (Canterbury, UK, Sept. 1984), North-Holland, Amsterdam.

[Feldman 79] Feldman, S., Make — A Program for Maintaining Computer Programs, Software Practice and Experience, Vol. 9, No. 3, Mar. 1979, pp. 255-265.

[Hood 86] Hood, R., Kennedy, K., Müller, H. A., Efficient Recompilation of Module Interfaces in a Software Development Environment, in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ed. P. Henderson, Palo Alto, California, Dec. 9-11, 1986, pp. 180-189.

[Järvinen 88] Järvinen, H-M., Principles of Intelligent Recompilation, Private communication.

[Liskov 86] Liskov, B., Guttag, J., Abstraction and Specification in Program Development, The MIT Press, Cambridge, Massachusetts, 1986

[Rain 84] Rain, M., Avoiding Trickle-down Recompilation in the Mary2 Implementation, Software Practice and Experience, Vol. 14, No. 12, Dec. 1984, pp. 1149-1157.

[Tichy 86] Tichy, W. F., Smart Recompilation, ACM TOPLAS, Vol. 8, No. 3, July 1986, pp. 273-291.