Combining Rule-Based and Procedural Programming in the XC and XE Programming Languages

Licentiate's Thesis

Esko Nuutila

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelytekniikan laitos

Helsinki University of Technology
Faculty of Information Technology
Department of Computer Science

Otaniemi 1990

TKO-A28
ISBN 951-22-0218-2
ISSN 0785-6644

TKK OFFSET 1990

Abstract

Rule-based programming is studied in the context of embedded expert systems. Problems in using traditional rule-based languages for embedded applications are pointed out. The problems are related to data representation, program structure, interaction with procedural code, program reliability, and to the RETE pattern matching algorithm that is used in most of these languages. A twofold solution to these problems is proposed. Firstly, rule-based and procedural code is combined in the same language by using abstract data types as the unifying component. Secondly, non state-saving pattern matching algorithms are used instead of the RETE algorithm. To test this solution three programming languages — XC, XD, and XE — have been designed. XC extends the C++ language by rules, working memories and rule sets. It is implemented using a preprocessing technique. XE is a new general purpose programming language that combines procedures, iterators, abstract data types, rules, and type parameters. A compiler has been implemented for XE. It is a part of an integrated programming environment. A smart recompilation algorithm is used in the compiler to avoid unnecessary compilations after modifications to the modules of an XE program. XD (not described in this thesis) is an implementation of the XE rule mechanism on top of C and C++. A few application programs have been written in XC and XE. According to the experience, the approach of combining rule-based and procedural programming in XC, XD, and XE is valid.

Foreword

The research presented in this thesis was pursued at the Helsinki University of Technology within the ExBed project during the years 1986 – 1989. The work was funded by the Technology Development Centre, Nokia Corporation, and KONE Corporation.

During the four years many people contributed to this work and I would like to thank them all. In particular I am in debt to the members of the ExBed project group: Markku Tamminen, Juha Kuusela, Jari Arkko, Vesa Hirvisalo, Jussi Rintanen, and Jukka Veilahti. The achievement documented here is largely theirs.

I wish to thank our industrial partners who directed the research with many useful comments and allowed us to use their corporations as test beds. In particular I wish to thank Nassim Bouteldja and Timo Lehtimäki, who contributed in an important way by being the first users of the XE language and the XE programming environment.

I also wish to thank professor Markku Syrjänen, the supervisor of my thesis, and Ora Lassila who read the manuscript. Their comments and constructive criticism had a major effect on the final form of this thesis.

Finally, I want to present my best thanks to my wife and to my children for their love and support during the project.

In a large and long-lasting project like the ExBed project, it is difficult to tell exactly who has actually done a particular part of the work. For example, the XE language, the XE programming environment and the metaprogramming tools were designed by the whole project group. However, there are some parts of the work that are clearly mine. I designed and implemented the XC language and the rule mechanism of the XE language. I am also responsible for the smart recompilation algorithm that is used in the XE compiler.

Otaniemi, March 1, 1990.

Esko Nuutila

Contents

1 Introduction

The ExBed project was founded to determine to which extent it is feasible to apply knowledge-based techniques within embedded systems, and what such techniques should look like. The resulting environment was to consist of two distinct parts: the development environment, where embedded applications are programmed and tested, and the actual embedded run-time environment. At run-time the resources of a "standard" embedded microcomputer provided a limiting factor for functionality, but there were no similar limitations on the development environment. The expert system paradigms supported by ExBed were to be integrated with a procedural host language.

The ExBed project developed a succession of three programming languages, XC, XD, and XE, each of which integrate production rules with a procedural programming language. This thesis describes the approach to rule-based programming that was used in the project. The use of abstract data types forms a basis to the integration of procedural and rule-based programming in the languages that were developed in the project. The XC and XE programming languages are described as well as the implementation of XE. XD is described in [56]. For other documentation of the results of the ExBed project see [51, 12, 11, 9, 10, 32, 15, 8, 5, 33, 40, 4, 6, 50, 7, 3, 2, 14, 39].

The outline of the thesis is as follows. In chapter 2 we describe the approach to rule-based programming that was taken in the project, namely the combination of rules and abstract data types. Chapter 3 presents the XC language, its implementation and the experiences that we have with XC. The XE language is described in chapter 4. Chapter 5 describes the implementation of the XE language and chapter 6 the smart recompilation algorithm that is used in the XE implementation. Conclusions and directions of further research are presented in chapter 7.

2 Rules and abstract data types

In this chapter we try to motivate the use of rule-based programming in embedded applications. We point out problems in traditional rule-based languages which make them unsuitable for embedded applications. We then describe the solutions to these problems that were developed in the ExBed project. Finally we present other work done in this area.

2.1 Applying rule-based programming to embedded applications

The rule-based paradigm is the most widely used implementation technique in knowledge-based software development. Its techniques and generally associated benefits are described in [31]. In the embedded systems area, most of the knowledge-based applications developed have used production rules as the main knowledge representation paradigm [19, 28, 37, 58, 13, 71, 72]. Since the work of Fagan, forward chaining rules are usually assumed most suitable for these systems.

The production rules are suitable for the implementation of knowledge-based embedded systems for the following reasons [14]:

- The data-driven inference strategy is rather simple and it corresponds to a general model of reacting to appropriate external stimuli, commonly used in embedded systems software development.
- It is often difficult to develop deep models of the knowledge associated with embedded applications. A realistic alternative is shallow knowledge, best represented with rules.
- Efficient implementations of rule-based systems are possible.
- Efficient development tools, like compilers, are possible.

2.2 Problems in using traditional rule-based languages

In the beginning of the project we had experience in implementing and using traditional “OPS5-like” [23] production system languages [41]. These languages have some properties that cause problems when applied to embedded applications.

2.2.1 Pattern matching

The RETE match algorithm developed by C. L. Forgy [22, 24] is widely believed to be the best algorithm for finding the set of instantiated rules in a production system. The algorithm derives its efficiency from two assumed features of production systems, namely: (1) structural similarity of the left-hand sides of the rules, and (2) temporal redundancy in the working memory. The latter assumption means that only a small number of working memory elements change from cycle to cycle. Under these assumptions the algorithm is rather fast because it never explicitly iterates over the working memory or the set of production rules. This is achieved by using a discrimination network compiled from the left-hand sides of the rules. The network "indexes" the rules by working memory elements. Partial matches of the left-hand sides are stored in the discrimination network while complete matches are stored in the conflict set. All modifications to the working memory are propagated through the network.

The RETE algorithm trades memory — both dynamic and static — for speed of execution. The amount of dynamic memory used can vary much during one execution. It is also difficult to predict the amount needed. Clearly, these are unwanted properties of an algorithm to be used for real-time control.

Modifications to the working memory are rather expensive in RETE, because they must be propagated through the network. Typically some hundreds of modifications per second can be made to the working memory. Thus, if the temporal redundancy assumption is false, the algorithm runs very slowly. Unfortunately this seems to be the case in many real-time applications. Often large amounts of data must be processed in a very short time.

Processing one modification to the working memory takes a rather long time. During this time the contents of the RETE network are inconsistent and the program should not be interrupted. Also this is undesirable in real-time applications.

2.2.2 Data representation

Production systems usually have some predefined and rather restricted way for representing working memory elements, i.e. the data used by the system. The classes of attribute — value pairs in OPS5 are one such example. It is rather obvious that no single fixed mechanism is suitable for all applications. Try, for example, to represent bit vectors with OPS5 classes.

Furthermore, the internal representation of working memory elements is often complicated and direct access to the data is impossible, because it could cause inconsistency in the saved state (network) of the RETE algorithm. This causes problems when a production system is interfaced with procedural code. In some systems this problem is solved by using multiple representations of data [41, 71]. This, however, requires conversions between the different representations and introduces a new problem, namely the maintenance of consistency between them.

2.2.3 Program structure

In non-AI applications the decomposition of programs into modules has long been the basic method for improving the flexibility, comprehensibility and maintainability of large systems [53, 44]. In rule-based systems this method has usually been neglected. Instead, these systems generally consist of one linear set of rules that communicate via a global database. This has caused problems when the systems have grown bigger.

2.2.4 Interaction with procedural software

Any software development project in embedded systems will also involve parts that are best coded using a procedural language. It is therefore important to be able to mix, as freely as possible, both paradigms in the same application.

Usually the time-critical parts have to be implemented in procedural languages while the rules are used in the implementation of higher-level, non-critical parts. The problem with languages that use the RETE algorithm is that the algorithm can only use data that is propagated through its network. Thus, the mere existence of the network can slow down the procedural parts of the program.

2.2.5 Reliability

Reliability requirements are very high in embedded systems. Often these systems operate continuously and there is no possibility for human interruption in the case of malfunction. On the other hand, the testing of the embedded software is often difficult, because it is usually developed in a hardware environment other than the target machine.

It would be beneficial to do as much verification as possible in the development environment. For example, the languages that are used in the development should support compile-time type checking. Traditionally (before OPS83 [25]) rule-based languages do not have strong typing.

2.3 Our approach

Our solution to the problems that were described above is twofold. Firstly, we have abandoned the RETE algorithm. Instead, we use non state-saving pattern matching algorithms. Secondly, we combine rule-based and procedural programming in the same language by using abstract data types as a unifying component.

2.3.1 Non state-saving pattern matching algorithms

To overcome the problems related to the RETE algorithm we have studied the use of non state-saving production system algorithms, i.e. algorithms that do not store any information about the partial matches and must therefore iterate over the working memory and the rules to find instantiated rules.

A non state-saving algorithm requires little static or dynamic memory, it does not slow down the procedural code like the RETE algorithm, and finally, the working memory operations are more fine-grained than in RETE and therefore the algorithm can be interrupted more rapidly.

An example of such an algorithm is one that scans the rules. At every rule it generates every possible combination of working memory elements and checks whether the left-hand side of the rule is satisfied. The complexity of this kind of an algorithm is terrible, if used carelessly.

The important observation we have made, however, is that such algorithms are not as inefficient as often claimed.

It is often possible to partition the rules into small sets that contain rules for different subtasks. Instead of scanning all the rules at every cycle of the inference process, only the relevant rule sets have to be processed.

The same kind of modularization can be applied to the data that the rules access. There is no need to use one global working memory that contains all data elements. Instead, several working memories that contain data of specific type can be used. The amount of possible combinations of data that satisfy the left-hand side of a rule reduces dramatically when only the relevant data elements are accessed. Often it is possible to use no working memories at all. The rules can access the the program variables directly as in Stimulus [57]. In this case the overheads of rule execution are very small.

The modularization of the rule set and data also makes the development and maintenance of the software easier.

By allowing problem specific optimizations the execution can be made even more efficient. For instance, the user can define own working memory data structures, e.g.

hash tables, according to the operations that are needed. The search for matching data elements can be restricted by using query optimization techniques. The user can also modify the rule instance search order.

2.3.2 Abstract data types

Abstract data types are currently the most important method in program design. Choosing the right data structures is crucial in achieving a clear and efficient program. In the absence of abstract data types, data structures must be defined too early — they must be specified before the implementations of the using modules can be designed. Abstract data types allow us to defer decisions about data structures until the uses of the data are fully understood. Abstract data types are also valuable during program modification and maintenance. In this phase, data structures are particularly likely to change, either to improve performance or to accommodate changing requirements. Abstract data types limit the changes to just the implementation of the type; none of the using modules need be changed [44]. For these reasons, most modern programming languages contain some kind of data abstraction facility, e.g., clusters in CLU [43] and classes in object-oriented languages like C++ [66] and Smalltalk [26].

Abstract data types form a very general mechanism. Bit vectors can be represented as abstract data types as neatly as attribute — value tuples that are used in the traditional production system languages. Thus, the use of abstract data types solves the problem of restricted data representation facilities related to the production system languages.

The implementation of the RETE algorithm is probably easier when simple OPS5-like working memory elements are used instead of abstract data types. However, the use of abstract data types does not cause any difficulties in the implementation of a non state-saving pattern matching algorithm.

2.3.3 Realizations

In order to test our ideas we have implemented three programming languages: XC, XD, and XE. XC is a rule-based extension of C++ [66]. XE is a new general-purpose programming language that combines abstract data types, procedures, iterators, and rule-based programming. XD is the implementation of the XE rule abstraction on top of C and C++. XC and XE are described in this thesis; for XD see [56].

2.4 Other work in the area

In this section we describe work that is related to ours. The main areas of interest are the rule pattern matching and the combination of rule-based and procedural programming. The reader is referred to the article [24] for a description of the original RETE algorithm.

2.4.1 Improvements to the RETE algorithm

Schor et al. describe several augmentations to the RETE algorithm that achieve improved performance and rule clarity [59].

In the original RETE algorithm, the working memory element modification operation is implemented by first removing the existing element from the working memory and then inserting the modified element back to the working memory. In the new RETE algorithm the modification operation is implemented as a single update-in-place operation. This results in much faster execution of working memory element modifications. It also prevents annoying re-triggering of rules that is typical to the original RETE algorithm.

In the new RETE network the condition elements can be arbitrarily grouped in order to share the two-input join nodes. The grouping mechanism also supports the negating of joined groups that was not possible in RETE.

The new algorithm also permits the sorting of partial matches and the selection of highest or lowest matching values.

A procedural match facility has been included in the new algorithm. In the normal RETE algorithm, all data that is used in the rules has to be matched agains the condition elements in the RETE net. This may cause unwanted triggering. The procedural match facility matches patterns only when the left-hand side of the rule is satisfied. It is implemented by turning on and off the nodes of the RETE network that correspond the procedural match in a rule when the rule is fired.

The new RETE algorithm also supports the addition of new rules on-the-fly.

The new algorithm is used in the YES/OPS production system language. Small projects have been done using the language. According the experience, the new algorithm runs orders of magnitudes faster than the old one in some cases. The performance improvements come from five factors: The modify as update-in-place simplifies the rules and thus reduces rule firings. The grouping construct allows more sharing of pattern tests. The sorted memory nodes reduce algorithm complexities. The procedural match reduces the number of active patterns. Finally, the new algorithm was timed and tuned carefully.

Ishida has studied the problem of how to automatically determine the best join structure for production system programs based on the statistics gathered from the previous executions of the program [35]. The evaluation results demonstrate that the presented algorithm generates more efficient programs than the ones that are manually optimized.

The efficiency of production systems rapidly decreases when the number of working memory elements becomes larger. This is because, in most implementations, the cost of join operations performed in the match process is directly proportional to the square of the number of working memory elements. Moreover, the inappropriate ordering of conditions causes a large amount of intermediate data, which increases the cost of subsequent join operations.

To cope with these problems, some production system languages [59, 13, 17] have introduced language facilities which enable users to specify an appropriate ordering or a clustering of join operations. There are three major heuristics: a) place restrictive conditions first, b) place volatile conditions last, and c) share join clusters among rules. These heuristics may be in conflict which each other. Thus, a method of trial and error is used. The algorithm that Ishida has designed removes the manual optimization task.

The algorithm is based on costs associated with the RETE network nodes. Associated with each node there are five parameters: token(n), memory(n), test(n), cost(n), and ratio(n). Before optimization, the production system is executed once and the values of parameters for one-input nodes are calculated. In the process of optimization, various join structures are created and the associated costs are evaluated. Various constraints are used in the creation of the candidate join structures in order to constrain the search.

The algorithm was evaluated by implementing an optimizer applicable to OPS5-like production systems. The optimizer reads a program and its execution statistics and outputs an optimized program. As a test case, a program for a real-world problem was given to the optimizer. It consisted of 107 rules, some of which contained more than 20 condition elements. The total number of inter-condition tests was reduced to 1/3 and the CPU time to 1/2. The result was better than by manual optimization.

Barachini and Theuretzbacher describe improvements on the RETE algorithm that make it more suitable for real-time applications [13]. The improvements have been implemented in the PAMELA production system language.

Barachini and Theuretzbacher have examined many AI-languages and decided that most of them were unable to handle time-critical problems efficiently. A pure production system is not fully applicable to process control applications, since asynchronous peripheral events may influence the recognize-act cycle during operation. Interrupt-handling facilities are fundamental for creating timely and elegant solutions to real-time problems. Being able to interrupt the recognize-act cycle and to modify an existing working memory element would be nice. However, in most production system languages it is impossible to handle a working memory element outside the scope of a rule. Further, the consistency of the RETE network cannot be guaranteed when a working memory element is changed during an interrupt.

To speed up the RETE algorithm, Barachini and Theuretzbacher have made several modifications to the RETE algorithm. The explicit token stacks have been removed. Instead, the nodes are represented as procedures that receive tokens as parameters (this is not a unique idea; it is presented also, for example, in [49]).

In the left and right memories of the two-input nodes, counters are associated with all tokens telling the number of consistently bound tokens in the other memory. In the memory following the two-input node, two pointers are stored for each token. The first pointer refers to the counter for the part of the token received from left and the other for the counter of the part received from right. When a token X with a negative tag enters a two-input node there is no need to match it against the tokens in the other memory and perform the tests. Instead, the memory following the two-input node is scanned for tokens with X as the left or the right part. Using the counters associated with the pointers the search for tokens that contain X as a part can be stopped when the counters decrement to zero.

Reorderings of the tests in the condition elements are used to reduce the number of nodes in the RETE network. These reorderings do not always lead to a faster execution time. Several levels of node sharing can be used in PAMELA.

Barachini and Theuretzbacher have performed a set of tests between PAMELA and OPS83 in order to test the efficiency. In some tasks PAMELA proved to be only twice as fast as OPS83. However, there were tasks where PAMELA was about 30 times faster than OPS83. Compared with the measurements in our XC paper [51] they claim that most criticisms of the RETE algorithm can be rejected.

In order to be able to handle interrupts, PAMELA contains new working memory operations SCAN (searching for a working memory element), Q_MAKE, Q_CHANGE, and Q_REMOVE. The Q operations are queued and performed only after the execution of the right-hand side of the rule is completed (or when a *synchronization slot* or a MAKE, CHANGE, or REMOVE operation is encountered). The language also contains a *DEMON-concept*. A demon is a rule instance that is fired immediately after it has matched the data that it requires. A separate demon conflict set is used. In addition, the language contains facilities that can be used in the right-hand side of the rule for determining whether a working memory element still exists and whether it has been modified by a demon.

Finally, Barachini and Theuretzbacher say that they believe that additional runtime optimizations to the ones that they and others have proposed can only be achieved by switching to parallel architectures.

Without any empirical experiments it is hard to tell exactly how much these improvements help in avoiding the problems that we described in section 2.2. However, the temporal redundancy assumption seems to be crucial for fast execution also in the improved RETE algorithms. The granularity of the working memory operations is still bigger than in non state-saving algorithms. The problems of reliability and data representation are not handled in a better way than in the original algorithm.

2.4.2 TREAT — an alternative to RETE

An alternative to the RETE algorithm, called TREAT, has been designed by Daniel Miranker [47].

TREAT does not save as much state between the cycles of the production system execution as the RETE algorithm. Specifically, it does not save the joins of the condition elements. Only the working memory elements that partially match condition elements of a production are saved. Associated with each condition element in the production system is a running count indicating the number of working memory elements partially matching the condition element. The counts are used in determining the set of *active* rules, i.e. rules that may have instantiations. When a working memory element is added to or removed from the working memory element, only the active rules have to be searched for instantiations.

In an empirical study [47], five OPS5 programs were used. Because the implementations of the TREAT and the RETE algorithms may be of different quality, the number of comparisons required to do variable bindings were used as the efficiency measure. TREAT outperformed RETE in all cases, often by more than fifty percent. This supports an unsubstantiated conjecture made by McDermott, Newell, and Moore, that the state saving mechanism employed in the RETE match, condition-element support, may not be worthwhile [45].

However, the study made by Nayak, Gupta, and Rosenbloom shows different results [48]. The RETE and TREAT algorithms were compared in the context of four different Soar programs. Using the number of tokens processed by each algorithm as the performance metric, it was shown that RETE performs better than TREAT in most cases.

It would be interesting to compare RETE and TREAT in the context of a language that combines rule-based and procedural programming. TREAT may avoid some of the problems that make RETE unsuitable for embedded and real-time applications.

2.4.3 Exploiting parallelism in production systems

Production systems, on the surface, appear to be capable of exploiting large amounts of parallelism — it is possible to match each rule to the data memory in parallel. In practice, however, the speed-up from parallelism is quite limited, less than 10-fold [29]. The reasons for the small speed-up are: (1) the small number of rules relevant to each change to data memory; (2) the large variation in processing required by the relevant rules; and (3) the small number of changes made to data memory between synchronization steps.

Various computer architectures have been designed for the parallel execution of rule-based systems. The match algorithms that are used in these machines are usually based on the RETE algorithm.

The *production system machine* [29] is a shared memory multiprocessor with about 32-64 processors. Each processor is a high performance computer with a small amount of private memory and a cache. The processors are connected to the shared memory via one or more shared busses. The machine exploits parallelism at a very fine granularity. The RETE network nodes are permitted to process more than one input token at a given time, and in some cases several tasks are run in parallel to handle the processing of a single token. The machine supports a hardware task scheduler to help enqueue the node activations that need to be evaluated and to help assign the pending node activations to idle processors.

DADO is a massively parallel tree-structured machine being developed at the Columbia University [65]. The prototype system has 1,023 node processors based on Intel 871 single chip computers. These processing elements contain 4K of EPROM and 256 bytes of on chip RAM. In addition, each processing node has 8K bytes of external RAM and a special purpose VLSI implemented switch to connect the node to its parents and children. Two algorithms offering the highest performance are the parallel RETE algorithm and the TREAT algorithm.

The NON-VON computer is another massively parallel tree-structured machine being developed at the Columbia University [61]. The proposed machine architecture consists of a very large number (16,000-1,000,000) of small processing elements, each with a small local memory (32-256 bytes). There are also some large processing elements with large local memories and disk access.

A set of simulations have been performed in order to compare the production system machine, the DADO machine, and the NON-VON machine [29]. Simulations of the production system machine architecture with 32 2-MIPS processors show that, on average, it is possible to keep about 16 processors busy, resulting in a performance of about 9,400 working memory element changes per second [29].

The performance of the prototype DADO (consisting of sixteen thousand 0.5 MIPS 8-bit processors) is predicted to be around 175 working memory element changes per second. [29].

The predicted performance of NON-VON (consisting of 32 32-bit large processing elements and 16,000 8-bit small processing elements, each executing at 3 MIPS) is around 2,000 working memory changes per second [29]. The better performance of NON-VON compared to DADO can partly be attributed to the fact that the NON-VON processing elements are six times faster than those of the DADO.

Based on these results, it can be concluded that a shared memory multiprocessor with a small number of processors seems to be a better solution than a massively parallel tree-structured architecture. This is probably caused by the nature of the RETE algorithm. It is difficult to get very many processors busy and doing useful work. On the other hand, much of the busy time of the DADO and NON-VON machines is spent communicating information [29].

Gupta and Tambe have studied the use of message passing computers for implementing production systems [30]. The early message passing computers such as the Cosmic Cube had a high network latency and a high overhead of message handling, which made them unsuitable for production system implementation. The recent advances in interconnection network technology and processing node design have made the message passing computers more interesting. The architecture that Gupta and Tambe propose is based on a concurrent implementation of a hash-table. It seems to be possible to gain execution speed-ups that are comparable to those achieved in the production system machine.

2.4.4 Other languages that combine rule-based and procedural programming

OPS83 is the latest and most powerful member of the OPS family of rule-based languages [25]. Like the other OPS languages also OPS83 is based on the RETE pattern matching algorithm. The OPS83 provides one of the fastest RETE based rule execution facilities commercially available.

Similarly to the other languages of the OPS family, only the record-like working memory elements can be used for the data representation in the rules. Also OPS83 uses a global working memory and a global RETE network compiled from the left-hand sides of the rules. OPS83 contains better facilities for controlling the conflict resolution than the other OPS languages.

The biggest difference between OPS83 and its predecessors is that OPS83 supports ordinary procedural programming paradigm as well as the rule-based programming paradigm. OPS83 is a strongly typed language in the style of Pascal.

OPS83 has efficient implementations in C, which also means an interface to C, that can be found in minicomputers, workstations and standard microcomputers. The programming environment contains facilities for examining the working memory, the conflict set, etc.

To summarize, OPS83 contains most of the weaknesses that were described in section 2.2. The version of the RETE algorithm that is used does not contain any of the improvements presented in section 2.4.1. No improvements to the data representation facilities have been made. The program can be split into modules, but only one RETE network is compiled. It contains all the rules and only one working memory can be used. The only improvements are the combination of procedural and rule-based programming, and the strong typing mechanism. In section 3.3.3 we compare OPS83 and our XC language.

YES/L1 (Yorktown Expert System Language One) [46] is a programming language that extends PL/I to include a rule capability and facilities for declaring, referencing, and manipulating the working memory.

The design of the language is based on the experiences that the Yorktown group had in implementing the YES/MVS real-time expert system in OPS5. They found many shortcomings in OPS5, e.g., no facilities for structuring the rule set, restrictive left-hand sides of rules, and poor performance. In YES/L1, they try to solve these problems.

The rule execution in YES/L1 is based on the RETE algorithm. Some of the extensions to the RETE that were developed for the YES/OPS language [46] (see section 2.4.1) have also been incorporated into YES/L1. For example, the procedural match mechanism that reduces the size of the RETE network is used. YES/L1 provides rule subroutines that can be used for partitioning the rules into smaller rule sets. Separate RETE networks are used for different rule subroutines.

To support real-time applications, YES/L1 provides facilities for inter-process communication, timed reminders, and waiting for an event. These facilities are built on top of the services provided by the underlying operating system (either MVS or VM). For example, it is possible to insert working memory elements into a working memory of a different process.

The YES/L1 compiler translates YES/L1 programs into PL/I source and generates a rule description file. The PL/I compiler creates an object module from the PL/I source. Rule description files and object modules from many compilations are combined to create a load module and a RETE network. The programming environment also provides facilities for stepping through rules and displaying the working memory, the conflict set, the rules fired, and the RETE network.

YES/L1 seems to be a more suitable language for embedded applications than OPS83. However, without any practical experience with the language it is difficult to say how well the problems that we presented in section 2.2 are avoided.

SPOCK [20] is an example of a language that combines rules and procedural programming in Lisp. SPOCK is oriented towards real-time expert systems, but the approach taken seems to be similar to the one that was used, for example, in EPSL ([49, 41]) that is not oriented towards real-time systems. The rule execution is based on the RETE algorithm. The rule patterns are fully compiled into Lisp which is then compiled by the Lisp compiler. The compiled code does its own garbage collection by recovering the list cells used in the local RETE memories. The rule execution is rather fast, but it is obvious that this system contains most of the drawbacks that we mentioned in section 2.2.

3 The XC language

In this section we describe the basic constructs of XC. This is only an informal description and some more advanced features of the language are omitted. We first describe C++, that XC extends. The constructs of XC are then described. Finally we describe the experiences that we have in using XC for writing application programs and the benchmarks and comparisons that we have made between XC and OPS83 in order to evaluate the ideas behind XC.

3.1 C++

C++ [66] is an object-oriented extension of the C programming language [38]. Except for minor details, C++ is a superset of C.

In addition to the facilities provided by C, C++ provides facilities for defining new types in the form of *classes*. A class definition consists of a set of data and function declarations, called *members*.

The programmer can define class hierarchies. First a base class is defined and then new classes are *derived* from it. The derived classes *inherit* the functions and data of the base class. Currently only single inheritance is possible. New functions and data can be defined in the derived class and the functions and data of the base class can be redefined. Inheritance facilitates code factoring, i.e. code to perform a particular task is found in only one place instead of being replicated in many places. It is believed that this eases the task of software maintenance [54].

C++ contains also many other (more or less usable) features that C does not have. The programmer can define an initialization function for a class that is called whenever an object of that class is created. Because C++ has no automatic memory management it is also possible to define a destructor function that is called when the object is deleted. The language provides also implicit type conversions, dynamic typing, operator overloading, in-line functions, symbolic constants, and default function arguments. The C++ type checking facilities are clearly better than those of C [66].

The object-oriented capabilities of C++ have given XC much of its expressive power and they have also eased the implementation of XC. However, it should be noted that any other language with sufficient data type definition facilities could have been used as a basis for a similar extension.

3.2 An overview of XC

Now we will explain how XC extends C++. The main facilities of XC are *working memories*, *rules*, and *rulesets*.

3.2.1 Working memories

A working memory is a collection of pointers to objects of specified types. The working memory type specification is

```
wm(T1,T2,...,Tn)
```

where `T1,T2,...,Tn` are type names. It can be used like any type name in C++, e.g., `int` or `char`.

Example 1:

```
/* Define some C++ classes */

class Ball { ... };
class Box { ... };
class Stick { ... };

/* Define a working memory local_wm
   that can contain objects of types
   Ball and Box, but not those of
   e.g. Stick. */

 wm(Ball, Box) local_wm;
```

There are two operations for accessing working memories: `assert` and `retract`. Assert is used for inserting objects into a working memory and retract is used for removing objects from a working memory. Note that there is no need for a `modify` operation because the RETE algorithm is not used.

Example 2:

```
/* A variable of type Ball. */
Ball b;

/* Insert the new object into
   working memory local_wm */
local_wm.assert(&b);
```

```
/* Remove it from local_wm */
local_wm.retract(&b);

/* A variable of type Stick. */
Stick s;

/* This is illegal. The compiler
   will detect a type error */
local_wm.assert(&s);
```

A working memory is implemented as a C++ class by the XC translator.

3.2.2 Rules and rulesets

An XC rule definition consists of a header and a body. The rule body contains the condition and the action of the rule. The condition is a C++ expression and the action is a C++ block statement. The rule header consists of the name of the rule and the rule parameters. A rule parameter is a variable that iterates over the objects of a given type in a given working memory. Different parameters can iterate over different working memories. The format for a rule parameter is:

```
<type> <variable> 'in' <working-memory>
```

Example 3:

```
/* Assume that type Ball has
   function size(). Then rule r1
   removes all Balls that are
   bigger than MAX_SIZE from
   local_wm. */

rule r1(Ball* b in local_wm) {
  when (b->size() > MAX_SIZE) {
    local_wm.retract(b);
  }
}
```

Note that the rules can also access objects directly without using any working memories. This is often the most efficient way.

In XC rules are divided into modules called *rulesets*. An XC ruleset is like a C++ class, but it can contain production rule declarations and definitions in addition to data and functions. This convention of distinguishing between declaration and definition is the same that is used with functions of a class in C++. Note the similarity with the definition/implementation concepts of Modula 2 and similar languages.

Example 4:

```
ruleset Blocks_World {
  wm(Stick, Ball, Box) local_wm;
    ...
  /* Declare rule Move_Ball */
  rule Move_Ball(Ball* b in local_wm);
  /* Define rule Take_Box */
  rule Take_Box(Box* bx in local_wm) {
    ...
  }
  ...
}
...
/* Define rule Move_Ball of ruleset
   Blocks_World */
rule Blocks_World::Move_Ball(Ball* b in
                             local_wm) {
  ...
}
```

Rulesets are translated into C++ classes by the XC compiler. Every rule is translated into two functions, one for testing its condition and the other for performing its action.

3.2.3 Ruleset execution

The XC compiler produces automatically an executive function called `run` for every ruleset. This routine executes the *match* — *conflict resolution* — *act* loop until no applicable rules are found or the function `halt` is called. Halt is a parameterless function that is also generated automatically for every ruleset.

In the first part of the loop — *match* — the executive function searches for rules whose condition part is satisfied. For each rule it repeatedly binds the parameters to all possible objects in the working memories and evaluates the condition of the rule.

The next phase — *conflict resolution* — depends on the strategy used. There exists a simple default strategy that selects the first satisfied rule to execution. Robertson [57] claims that this usually suffices in real-time systems. If this is not appropriate the user can supply a conflict resolution strategy of his own. This is accomplished by writing a function that compares rule instances. Every time a new rule instance is found it is compared with the best instance so far and, if it is better, the new instance is stored as the best instance.

Note that compared to the RETE algorithm we gain much in memory efficiency by not storing all rule instances and sorting them at conflict resolution.

In the *act* phase the action part of the best rule instance is executed and the loop starts again.

3.2.4 Derived rulesets

In XC a ruleset hierarchy can be constructed in the same way as a class hierarchy in C++. New and redefined rules, data and functions can be defined in the derived rulesets. Rules can also be removed by defining them in the derived ruleset with an empty body. The conflict resolution strategy is inherited by default, but it can also be redefined in the derived ruleset.

3.2.5 Embedding

The support for continuously operating real-time processes in XC resembles closely what has been offered by YES/MVS [28] and YES/L1 [46]. The communication between independent XC processes is based on the message passing services of the underlying operating system. Whenever a process is a ruleset its match — conflict resolution — act loop contains a phase during which external messages are recognized and transmitted to the working memories. The concept of time as such is not built into XC but is handled by library data types and procedures.

3.2.6 The XC compiler

The XC compiler is implemented as a preprocessor that translates XC definitions into C++ code which is then compiled by the C++ compiler. The preprocessor is a rather simple program that leaves as much work as possible to the C++ compiler. For example, all type checking is done by the C++ compiler. The phases of the compilation of a typical XC program are shown in figure 1.

3.3 Experiences with XC

In this subsection we describe the experiences that we have with XC. We have implemented two application programs in XC and compared XC with OPS83.

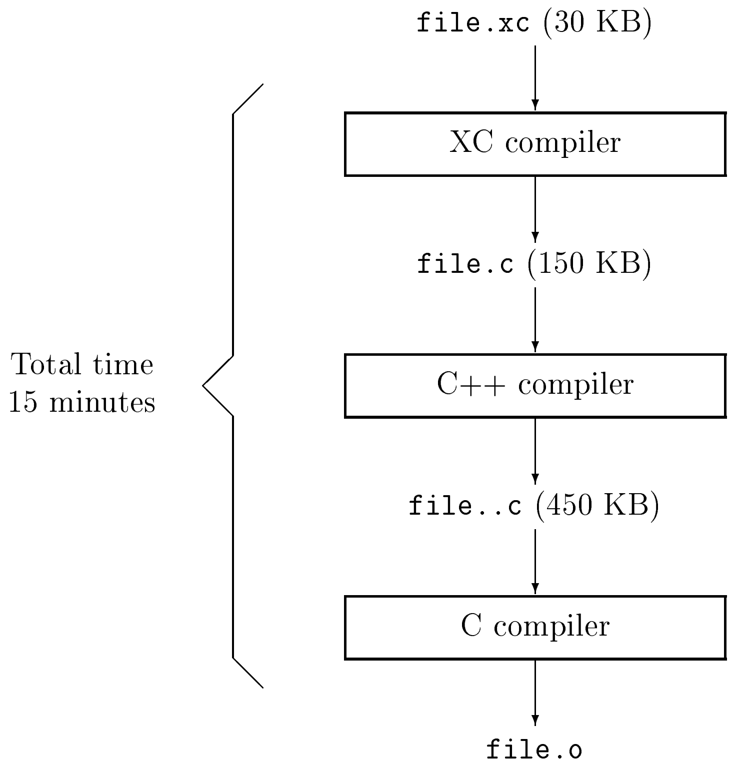

Figure 1: A typical XC compilation.

3.3.1 DPR

The first application prototype was a system for the demand pattern recognition of an elevator group. Based on the measured intensities of up and down traffic, DPR tries to infer what is the true type of the elevator traffic at each moment, i.e. is it "outgoing", "incoming", "up peak", "down peak", "two-way", or "mixed" traffic. Different control strategies can then be used based on the type of the traffic (see [3] for a more thorough description).

DPR was implemented in XC. The work was mainly performed in a Unix environment, and the operational result was later interfaced to KONE's group control system. The system made it possible to follow the traffic pattern floor by floor. Statistical monitoring and inference methods were used, together with rule based reasoning.

To make it possible to develop DPR at the university also a simulator of elevator systems was needed. Therefore the ExLift (Experimental Lift Group Simulator) system was constructed by using C++ and the OOPS system for Smalltalk-like object-oriented programming in C++ [27].

3.3.2 Filex

The Filex file system expert [11] monitors the use of file space and provides the users with services for managing their files and for freeing disk space. Filex contains about 40 inference rules.

Filex is run every night. The rules of the system first try to infer whether a file is worth keeping on disk by trying to determine the nature of the file, i.e. the information content of the file, the probability that the file will be read again, and the size of the file. Here is an example of a Filex rule that should annihilate `core` files:

```
rule core()
{
  when(streql(basename(filename), "core")) {
    Information_content = 0.0;
    Usage_probability = 0.0;
  }
}
```

The information inferred by the rules is then considered in light of the amount of free space in the system. For each file, Filex reaches a recommendation on what should be done with it. This recommendation is presented to the owner of the file next time he or she logs in. In extreme cases, Filex can remove files without authorization from the owner of the files.

XC was found to be a good tool for building Filex. The close relationship between C++ and XC enabled the use of the whole C++ language in rules; e.g., normal C++ expressions in the left-hand sides. The possibility of using complex problem-specific data structures with rules was found important. The types of working memory elements found in, e.g., OPS5 simply do not cover the needs of all rule based systems.

The simplicity of XC was found valuable, because simple rules could be written for simple situations without extra overhead from complex rule and working memory definitions. However, it was also found out that some features needed were missing from XC. The worst problem was the lack of a refraction mechanism, i.e., a mechanism for disabling the rules from firing again with the same data. This drawback follows from the choice of not using the RETE algorithm. In Filex this problem was solved with a flag for every rule, but this is not always possible.

Since the volume of data is huge in file systems, speed of execution was a concern in the Filex system. With the simple set of rules and minimal amount of working memory elements XC offered a comfortable efficiency.

The program development environment of XC was a big disappointment. The test cycle was very long as there were many compilation phases, a whole rule set had to be compiled as a unit, and there was no dynamic linking. The C++ compiler that was used contained many annoying errors. Currently there are better tools available for C++ program development.

3.3.3 Comparing XC with OPS83

In addition of implementing application programs with XC we compared it with OPS83 in order to test the XC design hypotheses. An application program was written in OPS83 to have experiences with the language. We also ran a set of benchmarks with OPS83 and XC in order to compare their efficiency.

DXPERT [14] is an expert system for the interpretation of alarm messages originating from the DX 200 digital switching system. It can analyze an alarm dump and provide a fault diagnostic together with recommendations on remedial action. It is implemented using OPS83 on an MS-DOS PC loosely connected to the switch through a serial communications link. The system contained approximately 100 OPS83 rules with an average of 5 condition patterns per rule. The conclusions resulting from work with this application can be summarized as follows:

- The importance of the availability of both the procedural and rule-based paradigm in the same language was confirmed.
- Tailorability of the inference engine to the application, as afforded by OPS83 and XC, is a must. The standard conflict resolution strategies like MEA found in OPS5 [16] have an important flaw: they are unaware of the characteristics of the application environment.
- The absence of a modularity mechanism for rules in OPS83 was a drawback.
- The strict type checking of OPS83 was very helpful in preventing certain kinds of mistakes. Nevertheless, the absence of an integrated programming environment providing support for recompilation, debugging and version control was felt.

To test the efficiency of XC it was benchmarked agains OPS83 using a couple of small examples. The example problems were the Towers of Hanoi and Balls and Boxes problems of various sizes (fitting balls with matching attributes into boxes). In these tests XC run 5 to 23 times faster than OPS83 depending on the application. OPS83 also used 50 to 1,000 percent more memory than XC [51].

4 The XE language

Based on the experiences with XC, data abstraction was found to be a suitable basis for implementing embedded expert systems. The main enhancement required for rule based programming was yet increased openness. That is, the programmer was to be given complete freedom in the design of the data structures and algorithms related to both rule sets and working memories. We also preferred parameterization to inheritance as a type generalization mechanism in many cases.

As explained in the previous chapter, the programming environment of XC was a big disappointment to people used to a Lisp environment. The program development cycle was very long as there were many compilation phases, a whole rule set had to be compiled as a unit, and there was no dynamic linking. Further there was no debugging support for C++ at that time. Also, programs written in XC would have been very difficult to analyze automatically, because XC contains as a subset C++ — and therefore also C. All this led to a reconsideration of the language strategy.

It was decided to base the second phase of the project on a more ambitious approach than extending an existing language via precompilation technique. We decided to design a general purpose programming language XE, which would support both rule based programming and parameterized data abstractions. Because the CLU language satisfied the latter objective it was taken as a model for the design of XE. However, XE is not an extension of CLU; instead it is an integral whole specified from its own premises.

In the design of XE it was considered especially important that the XE compiler would be a part of an integrated programming environment that would support programming in XE in the same way as a Lisp environment supports programming in Lisp [70].

In this chapter we give an overview of XE. The implementation of XE is described in the next chapters. A more complete description of XE can be found in [8] and in [5].

4.1 An overview of XE

XE is a general purpose programming language that supports the use of abstractions in program construction. Beside the procedure abstraction available in all programming languages and the iterator and data abstractions supported by CLU, XE provides the programmer an integrated rule abstraction. XE also contains structures that are useful in the development of large embedded systems, e.g., mechanisms for calling routines written in another languages and allocating objects in static memory.

4.1.1 Modules

An XE program is made up of one or more modules. Each module is a *procedure*, *iterator*, *rule*, *data type*, or *constant*. Procedures, iterators, and rules are called *routines*, because they can be invoked. There exists no concept of a main program, because any routine that can be directly invoked by the execution environment can act as a main program.

A *procedure* is used for implementing a procedural abstraction, i.e., a mapping from a set of argument objects to a set of result objects. A procedure is provided with zero or more argument objects, it returns zero or more result objects, and possibly modifies the state of its arguments. Procedures can terminate exceptionally signaling a name and zero or more signal objects.

An *iterator* implements an iteration abstraction. One can think of the iterator as a control abstraction generalizing the `for` construct found in current programming languages, which can iterate over ranges of integers. The XE `for` construct can iterate over collections of any type of object. The iterator yields a sequence of *items*, one item at a time, and the `for` construct consumes them. An item is a group of zero or more objects. As procedures, iterators are given zero or more argument objects. They can terminate in a normal way returning nothing, or exceptionally in the same way as procedures. Iterators are central to the rule concept, as embodied in XE.

A *rule* implements a forward chaining rule abstraction. A rule is a special kind of iterator. It is given zero or more argument objects and it yields a sequence of items, that it yields one at a time. Usually they contain data that satisfies the *condition* of the rule. Rules are handled more thoroughly in the next subsection.

A *data type* implements a data abstraction. It defines a set of objects and a set of primitive operations to create, examine and manipulate them. The operations can be procedures, iterators, or rules. The data types of XE are called abstract because the concrete representation of an object can be seen only by routines inside the definition of the data type. In this approach the capabilities of an object and the applicability of abstractions is defined, not by membership in a type hierarchy, but by the external interface of the object and the interface requirements of the abstractions. A typical data type is `int` whose objects belong to a subrange of the mathematical integers and whose operations form the usual integer arithmetics.

Unlike in CLU, the XE modules 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 module can be bound by the nested module. Thus it is possible to write lexical closures in XE.

Each module has a *header* that specifies the interface of the module and a *body* containing the implementation. The implementation is not visible to other modules. As in CLU, no special interface or specification modules are used.

The *body* of a routine definition consists of a sequence of statements and can also contain declarations of local variables. The statements in the body can refer only to the local variables, the (nested) modules and the formal names declared in the header. The body of a module consists of expressions, statements and definitions of other modules.

4.1.2 Parameterization

As in CLU, all XE modules, except constants, can be parameterized. This allows a set of related abstractions to be defined by a single module. A parameterized module is called a *generator*, because it defines an ordinary module each time it is provided with legal actual parameters; for example `array[int]` and `array[string]`. The modules that are created this way are called *instances* of the generator.

It is possible to place requirements on the parameter types, for example, that the parameter type of an ordered set generator should have a less-than operator.

Sometimes a potential parameter type does not have a required operation, but it is possible to implement it using the existing operations. In CLU this kind of problem cannot be solved, but in XE the `use` construct can be used for inserting the missing operations to the type when creating an instance.

We wanted to write all the XE basic types and type generators in XE. The simple parameterization abstraction of CLU was found to be too restricted for this. In CLU it is not possible to write type generators such as `record` even though this kind of generators are included in the built-in library of the language. In XE the user can write *selector generators* that are parameterized by a field list. A field list is a list of name-type pairs, such as in `record{name: string, age: int}`.

4.1.3 Objects and identifiers

XE programs refer to *objects* and perform computations on them. Each object belongs to a data type. The data type completely determines what kinds of values an object may have and what operations can be performed on it. For example, an `add` operation is meaningful for integer objects but not for string objects.

To operate on objects, programs need a way to refer to them, and to do this they use *identifiers*. The identifiers that may refer to different objects as time goes by are called *variables*. A variable always has a type. The type checking of XE guarantees that a variable always refers to an object of an allowed type. Identifiers are *not* objects, they cannot be referred to and they do not contain anything; they are just labels for naming objects.

Objects come into existence though execution of various operations and exist within an *object universe*. For example, the `create` operation for arrays causes a new array object to be created in the object universe. Objects cannot be explicitly removed from the universe so that they can be thought of as existing forever. However, in practice, the space used by inaccessible objects is reclaimed by *garbage collection*. An object is *accessible* if it is referred to by an identifier or another accessible object.

Each object is distinct from all others. Usually an object cannot physically contain other objects, but it can contain references to other objects; when this happens, we say that the containing object *refers to* the object contained. Because objects can refer to other objects, it is possible to have cyclic objects (which refer to themselves) and recursive data structures without the use of explicit pointers. Also, it is possible for several objects to refer to the same object. When this happens, we say that they *share* the object referred to.

4.1.4 Assignment and invocation

There are two fundamental actions that relate to objects — assignment and invocation. Assigning an object to a variable gives a name to the object. After the assignment, the variable can be used for referring to the object. A variable can refer only to one object at a time; the old reference (if any) disappears when an object is assigned to the variable.

In an assignment the object is not copied; an object can have several names, i.e., the variables sharing it. Assignment does not affect the value of the object assigned; the object itself remains untouched.

Several forms of assignment are used in XE. Besides the simple assignment where the value of an expression is assigned to a variable the language contains two forms of multiple assignment. Some forms of *syntactic sugar* (see section 4.1.6) can be used in the left-hand side of the assignment.

In addition to explicit assignment XE has a number of constructs performing *implicit* assignment. They are the argument passing mechanism, that is discussed below, and the `signal`, `exit`, `return`, and `yield` statements. All assignments can operate on multiple values that are given either by a list of expressions or by a single invocation.

The other fundamental action called invocation has the form

```
primary( argument_expression, ... )
```

The primary is a restricted form of an expression; usually it is just the name of a routine. The argument expressions are arbitrary expressions. The sequence of activities in performing an invocation are:

- The primary is evaluated and the result must be a routine, i.e., an iterator, a procedure, or a rule.
- The argument expressions are evaluated.
- New variables corresponding to the arguments are introduced.
- The values of the argument expressions are assigned to the variables corresponding the arguments (the types of the values much match the types of the arguments).
- Control is transferred to the routine.

When the invocation is terminated, control returns to the invoker and the result objects are available. If the invocation terminates exceptionally control will not go to the point of return; it is given to an exception handler.

4.1.5 Types

The concept of a data type is fundamental to XE. Each data type in XE is a data abstraction. Thus a data type defines a set of objects and a set of primitive operations to create, examine and manipulate those objects. XE provides a set of built-in types (such as `bool`), built-in type generators (such as `record`), and a mechanism that allows the user to define new types and type generators.

A data type is implemented by a *type module* that describes a *concrete representation* for objects of that data type and routines which implement its operations. The concrete representation is defined in terms of other data types. The data types of XE are called abstract because the concrete representation of an object can be seen only by routines inside the definition of the data type. Objects can be manipulated only via operations of their type; thus the operations of the type completely define the abstract behavior of an object.

One goal of XE is to make user defined types powerful. Built-in and user defined types are treated as uniformly as possible. Just as built-in types, user defined types can be parameterized and their operations can be freely declared to have parameters, arguments, return values, restrictions and exceptional terminating states. User defined types can also benefit from the *syntactic sugar* of XE, i.e., abbreviations.

4.1.6 Expressions and syntactic sugar

An XE expression evaluates to an object that is called the *result* or the *value* of the expression. Simple expressions, such as literals, variables, parameters, and routine names directly name their result object. More complex expressions are generally built up by using nested invocations of procedures. The value returned by the outermost invocation is the result of such an expression.

XE has prefix and infix operators for the common arithmetic and comparison operations and allows indexing and component selection using the normal notation:

```
a[i]
r.s
```

All these operation forms are syntactic sugar for canonical invocations. For example, the previous indexing and component selection expressions are syntactic sugar for the following invocations:

```
ta$fetch(a, i)     % ta is the type of a
tr$get_s(r)        % tr is the type of r
```

There are only five operations that correspond to no canonical invocation: the conditional operators `cand` and `cor`, the representation type conversion operations `up` and `down`, and the iteration termination predicate `terminated`.

4.1.7 Statements

The following conventional statements are available: the procedure invocation statement, the assignment statement, the `if` statement, the `case` statement, the `while` statement, the `break` statement for jumping out of a loop, the `continue` statement for jumping to the next iteration of a loop, and the `return` statement for returning control and values from a procedure.

Iterators and rules are invoked with the `for` and the `iterate` statements, and values are passed from an iterator or a rule to the `for` or `iterate` statement with the `yield` statement.

The `tagcase` statement can be used for handling variant records and similar constructions.

Exceptional conditions are raised with the `signal` and the `exit` statements, forwarded with the `resignal` statement, and handled with the `except` statement.

4.1.8 Embedding

As in XC, the communication between independent XC processes is based on the message passing services of the underlying operating system.

In order to make possible the interfacing of XE programs with existing software written in other languages, a foreign procedure call mechanism has been included in XE. The major problem with such foreign procedures is the argument and return value passing because of different representations of data. In XE the argument objects are converted by `dump` operations of their types before passing them to the foreign procedure invoked. The result objects produced by the foreign procedure are converted into the XE representation by invoking `undump` operations of their XE types. Thus it is possible to use all types that have the `dump` and `undump` operations (including user-defined types) in the communication with the outside world. These operations designed for efficient communication of binary data are provided for all built-in XE types and type generators.

One of the problems that has hampered the use of knowledge-based techniques in real-time embedded systems is the prohibitive time spent by the software on garbage collection. Using the `static` construct in XE, it is possible to define static objects that will not be subject to garbage collection.

4.1.9 The XE libraries

Libraries have a very important role in XE. The libraries actually form a part of the language. Writing own library modules and reusing code stored in libraries is typical for programming in XE.

The built-in types and type generators of XE are included in the basic library [8]. The built-in types are `null`, `bool`, `byte`, `int`, `word`, `char`, and `string`. The built-in type generators are `array`, `sequence` (immutable arrays), `tag` (enumeration types), `variant` (mutable variant records), `oneof` (immutable variant records), `record`, and `struct` (immutable records). The collection of built-ins could be easily changed.

The XE utility library [33] contains lots of useful abstract data types and type generators, such as lists, queues, stacks, sets, associative memories, and so on.

All XE libraries, including the basic library are written in XE. Even the record-like generators are implemented in XE using the field parameter list construct. The compiler knows about some basic operations of the built-in types. For example, it knows the basic arithmetic operations and memory access operations that are used in the implementation of the arrays, records, etc.

4.1.10 An example program

In order to make the reader more familiar with the XE language, we present a simple user-defined type generator and a piece of code that instantiates the generator and uses the instantiated type. Note that '%' starts a comment. An operation of a data type can be denoted either by giving the name of the type and the name of the operation separated by a dollar sign '$' or by using only the operation name and placing a colon sign ':' before an argument that is of the correct type; the latter is syntactic sugar for the former.

```
% Stack parameterized with maximal size and type
% of elements.
stack = datatype[maxn: int, t: type]
  is new, pop, push, elements
     % We put a restriction on element type: it should
     % have a 'default' operation.
     where t has default: proctype() returns(t)

  % The representation of a stack; not visible outside
  rep = record{n: int;
               a: array[t]}

  % 'New' creates a new empty stack object. It calls
  % the record constructor and the required default
  % operation.
  new = proc() returns(cvt)
    return({n: 0, a: array[t]$fill(maxn, t$default())})
  end

  % 'Pop' removes and returns the top most element of
  % stack or signals 'empty'. The argument type 'cvt'
  % means that the concrete representation of argument
  % 's' is used inside the procedure.
  pop = proc(s: cvt) returns(t) signals(empty)
    if s.n = 0 then signal(empty) end   % raise an exception
    s.n := s.n - 1
    return(s.a[n])
  end

  % 'Push' inserts a new element on top of stack or
  % signals 'full'.
  push = proc(s: cvt, e: t) signals(full)
    if s.n = maxn then signal(full) end
    s.a[n] := e
```

```
    s.n = s.n + 1
  end

  % 'Elements' is an iterator that yields the elements of
  % the stack one after another. It uses the 'elements'
  % iterator of the array type to get the elements of the array.
  elements = iter(s: cvt) yields(t)
    for e: t in elements(:s.a)
      yield(e)
    end
  end
end

% Example of instantiating and using the stack abstraction.
a_stack = stack[100, int use default = int$maxint]
as: a_stack := a_stack$new()
for i: int in [1..10] do
  push(:as, i)
end
sum: int := 0
for i: int in elements(:as) do
  sum := sum + 1
end
```

4.2 Rules and rule-based programming in XE

In XE, a rule is a special kind of iterator that computes a sequence of *rule instantiations.* A rule instantiation is a tuple of arbitrary XE objects. Usually it contains data that satisfies the condition of a rule and an action that should be applied to the data. It may also contain data that is used when comparing instantiations.

A rule is of the form:

rule ::= `rule` [parms] args [yields] [signals] [binds]
{ equate or own variable declarations}
when
conditions
:
[statements]
end

Each rule has a fixed number of arguments and it can be parameterized just like a procedure or an iterator. The *yields* declaration in the header declares the number, order, and types of the components of instantiations. The rest of a rule consists of a *condition*, and a *body*, which is a statement. Instantiations of the rule are created by **yield** statements in the body.

A condition is either a clause or a sequence of nested **for**-iterators, qualified by a clause:

condition	::=	`[` **for** [decl, ...] **in** invocation `]` condition
	|	clause
clause	::=	clause **and** clause
	|	clause **or** clause
	|	**not** clause
	|	(clause)
	|	`[` **some** [decl, ...] **in** invocation `]` clause
	|	`[` **all** [decl, ...] **in** invocation `]` clause
	|	predicate

A predicate is an expression of type `bool`. Complicated clauses can be built by using operators **and**, **or**, and **not**. Clauses can be quantified with **all** (universal) and **some** (existential) quantifiers. A *quantification* consists of a quantifier, a list of variable declarations, and an iterator invocation.

Rule invocation is performed as follows. (Here it may be helpful to inspect the examples below.) The actual argument objects are assigned to the formal arguments of the rule. The outermost **for**-iterator (if any) is invoked. If it yields something, the objects are assigned to the declared variables. Nested **for**-iterators are executed in the same way, except that when one of them terminates the closest surrounding iterator is resumed. If the innermost **for** iterator yields an item the clause contained in the condition is evaluated. If its value is **true** the expressions on the right-hand side are evaluated, the corresponding instantiation is yielded and the rule is temporarily suspended. (At this point the program can use the instantiation, e.g., fire it.) If the value is **false** the innermost **for** iterator is resumed and the clause is re-evaluated with the new variable bindings. When the rule is resumed the suspended iterator of the innermost **for** quantification is resumed.

The following rule yields its arguments (i, j) if $i > j$ and otherwise nothing.

```
trivial = rule(i, j: int) yields(int, int)
  when i > j:
    yield(i, j)
end
```

The following rule yields three instantiations. The first contains integer 1 and an action to output “2” and “1”; the next one integer 2 and an action to output “3” and “1”; finally integer 1 and an action to output “3” and “2”.

```
simple = rule(output: stream) yields(int, proctype())
  when [for i: int in [1 .. 3]]
         [for j: int in [1 .. 3]]
            i > j:
  yield(i - j,
        proc() binds(i, j)
          putl(:output, i)
          putl(:output, j)
        end)
end
```

Finally we present a rule with quantification. The following rule is taken from a file system expert program. It detects files residing in a directory called ”temp” or ”tmp”, or in their subdirectories.

```
rule_0012 = rule(f: filename) yields(string)
  when [some dir: string in components(:f)]
          dir = "temp" or
          dir = "tmp":
    yield("File is in temporary storage")
end
```

The rule abstraction of XE does not contain concepts such as rule set, working memory, conflict resolution, rule firing, and inference engine, which are typically used in rule-based programming. All these concepts can easily be built in XE itself, using the other abstractions, typically as follows:

- **Rule set**: An abstract data type whose concrete representation includes the rules belonging to the set. Amongst the operations provided by the data type, one or more iterator abstractions that yield the rules in the orders wanted would be included.
- **Working memory**: An abstract data type providing operations to put and retract XE objects. If a working memory is to be used in the same way as in pattern matching supported by OPS5 and similar inference engines, it would normally include one or more iterator abstractions to access relevant elements. The working memory could be made generic by using type parameterization. The type of the XE objects to be manipulated would then be supplied as a parameter when the working memory is first created. The rules can use several working memories. The rules can also access objects directly without any working memories.

- **Conflict resolution**: A conventional procedure abstraction that compares two rule instantiations and chooses the best one according to the strategy coded.
- **Rule firing**: An invocation of an action yielded by a rule instantiation.
- **Inference engine**: A procedure abstraction that iterates over rules in a rule set and over their instantiations. The order of rule iteration can be customized as wished during the rule set execution.

The following example is a minimal illustration of some of these concepts. Assume a parameterized datatype `stack` with a procedure `push` and an iterator `elements` (like the one that we presented in the previous section). First we define type constants `rtype` and `rstack`. Then we create `rs` (rule set), a stack of rules having the same type as `simple` above. Next we push `simple` and other rules of the same type into the stack of rules. After that, we iterate over all rules and their instantiations (inference engine), in order to find the instantiation with the highest value (conflict resolution) of the integer attribute yielded by the rule. Finally we execute (rule firing) the procedure (action part of rule) corresponding to the highest integer value found.

```
rtype = ruletype() yields(int, proctype())
rstack = stack[rtype]
rs: rstack := rstack$new()
imax: int := 0
pbest: proctype() := proc() end
push(:rs, simple)
...
for r: rtype in elements(:rs) do
  for i: int, p: proctype() in r() do
    if i > imax then
      imax, pbest := i, p
    end
  end
end
pbest()
```

4.3 Experiences with XE

We have gained experience with XE in two ways. Firstly, the XE libraries are written in XE. The libraries consist of 11,000 lines of XE code, divided into 32 top level data types, 26 procedures, two iterators, and 35 constants (the XE run-time system that was partly implemented in XE is not included). Writing the whole XE library in

XE itself simplified the implementation greatly. It also revealed lots of errors in the XE implementation. On the other hand, some of the facilities in XE have only been used in the implementation of the libraries. Thus, the language grew bigger because of this design decision.

Secondly, XEDA, a medium-sized application program was implemented in XE. XEDA (XE Diagnostic Advisor) is a diagnostic knowledge based system embedded into the DX 200 digital switch. It runs as a process under a real-time operating system. Communication with other processes involved in the maintenance function of the switch, is achieved trough asynchronous message passing.

The role of XEDA is to determine, through a knowledge-based analysis of the alarm history, the optimal sequencing of diagnostics tests. The optimal order is one in which the test program identifying the fault is run as early as possible. The need for this intelligent diagnostic arises from the fact that some tests can be very lengthy (30-60 minutes).

XEDA's structure reflects a methodology supported by XE where problem decomposition is based on the recognition of abstractions. The abstractions built in XEDA correspond to the objects involved in its domain area (DX files, alarms, diagnostics tests, etc.), together with control logic that uses the capabilities of these objects to solve the problem at hand. XEDA uses rule sets which are implemented as XE abstract data types. The working memory is a parameterized data type.

The inference engine is a procedure that includes in its cycle, a phase where it checks for any incoming messages. This is necessary, because the inferencing can last more than the specified time for responding to the supervision process of DX200. State-of-the-art production system languages, like OPS83 [25] do not offer this facility and could therefore not be used in an environment like DX200. Barachini and Theuretzbacher [13] claim that interrupt-handling facilities are fundamental for creating timely and elegant solutions to real-time problems. In XE, it is easy to implement these kinds of facilities, but in most production system languages this is impossible.

The XE language and the XE programming environment were found extremely helpful. A simple simulator for DX200 system calls was built quickly in XE itself on the Lisp machine. The strong type checking minimized error debugging on the target machine.

True to its name XEDA was completely programmed in XE, except for one PL/M module. Depending on the version (there are two versions, one for each type of switch) XEDA contains 100 — 200 rules, and altogether 5,000 — 7,000 lines of XE code. It was a major challenge to make XEDA fit into the mere 128 KB allotted to it in the DX 200 memory, and this challenge gave rise to many of the optimizations in the XE compiler. The code generated by the XE compiler is efficient. On average, 350 bytes of code was generated for each additional rule. This is 150 bytes less than what OPS83 would generate.

See [15] for a more extensive discussion of the use of XE in XEDA.

5 The XE implementation

The XE compiler is not an independent program like, for example, the C-compiler in the UNIX-environment. The parts of the XE compiler have been integrated into an uniform programming environment, together with an editor, a version management facility, an error handling facility, and various other program development facilities. The programming environment has been implemented in Common Lisp, using XE metaprogramming tools, and it uses a Symbolics workstation as its basis.

In this section we describe the XE compiler and the metaprogramming tools that have been used in its implementation. The other parts of the XE programming environment are described in [39].

5.1 Metaprogramming tools

The XE compiler was written in Common Lisp because we had good experience in using a Lisp machine in the rapid development of prototype systems. However, plain Common Lisp has some drawbacks that cause problems when it is used in the development of large systems. It also lacks some facilities that would be beneficial in the implementation of a compiler.

The main problem with Lisp is that it is not strongly typed. When a large program is being implemented by several programmers one can be sure that much time will be spent tracking down "deletion errors", i.e., missing access forms in an expression, and other type errors that would be detected at compilation time when using a strongly typed language. In many cases these kinds of errors will not be correctly detected even at run-time. In the Symbolics Common Lisp, for example, if we define two struct types A and B with `defstruct` and use an object of type A as an object of type B, no error will be detected if the indices of the fields that we access are not greater or equal to the number of fields of A.

A compiler has usually a large number of data structures. However, these structures can be divided into a small number of categories that have much in common. For example, there may be many types of abstract syntax tree nodes, but they all have some common structure and some common functions that may be applied on them. Defining these data structures and corresponding functions directly with the facilities of Common Lisp, e.g., with `defstruct` and `defun`, would be rather tedious. Also, if `defstructs` were used, we would always have to find out the type of the node, e.g., with `case`, before we know how to access it. Inevitable modifications of the data structure definitions would cause changes to many places in the compiler.

To overcome these problems we decided to implement three specialized "small languages" — **ROCC**, **AST**, and **DG** — on top of Common Lisp.

5.1.1 ROCC

ROCC (RObust Compiler-Compiler) [6] is an LALR(1) parser generator [50] for a Lisp environment. A grammar expressed as a Lisp macro is transformed to a parser. This parser is a function that can have arguments and return values. One may have several parsers in the same environment, and several instances of the same parser running at the same time.

Functionally ROCC is equivalent to YACC [1] except that it has a set of error correcting and recovery features. These features are optional and the user can define most of the recovery procedures himself. However, ROCC does offer a) information about the error and the state it occurred in and b) a framework for correcting errors and recovering from them.

The following example defines a parser for simple expressions made of numbers, '+' and '*' operations and parentheses. First the terminal symbols and their precedences are defined like in YACC and then the productions of the syntax are listed. The definition will expand to a function `expr-parser` that takes a lexical scanner function as an argument and returns the constructed expression.

```
(defparser expr-parser ()
  :token (id num leftpar rightpar)
  :assocs ((:left plus)
           (:left mul))
  (E (E plus E              (list '+ v1 v2))
     (E mul E               (list '* v1 v2))
     (leftpar E rightpar    (identity v1))
     (num                   (identity v1))))
```

5.1.2 AST

AST [50] (Abstract Syntax Tool) is an object oriented language for the concise definition of abstract syntaxes. Abstract syntax tree nodes are objects and methods can be applied to these objects. The structure of the abstract syntax defines a simple three level inheritance hierarchy.

An AST abstract syntax consists of a set of *productions*. A production defines a *domain* and a set of possible *options* of that domain. The options consist of a *name* and a sequence of *fields*. Every field has a name and a type. The type is either *domain*, `&opt` *domain*, `&list` *domain*, or `&list1` *domain*. If it is *domain* the field must have a value and the value must be a node of *domain*. If it is `&opt` *domain*, if the value is given it must be a node of *domain*. If it is `&list` *domain* the value must be a list of nodes of *domain*. If it is `&list1` *domain* the value must be a non-empty list of nodes of *domain*. There may also be &aux fields that may contain data that is not of any abstract syntax domain, e.g., attribute values.

Following is a simple example of an AST definition:

```
(define-abstract-syntax syn
  :productions
  ((stmt (:if (condition expr) (then stmt) (else &opt stmt))
         (:begin-end (body &list stmt))
         (:assign (var idn) (value expr)))
   (expr (:add (op1 expr) (op2 expr))
         (:var (var idn))
         (:int &aux (int integer))
         &aux (value integer))
   (idn (&aux (name string)))))
```

This defines an abstract syntax called syn that has three domains, stmt, expr, and idn. The first production that defines domain stmt consists of three options. The first option, with the name :if, has three fields with names condition, then, and else. The types of the fields are expr, stmt, and &opt stmt. The &opt definition means that the else field may be missing. The value of the field body of the second option (:begin-end) must be a list of nodes of domain stmt. In the production that defines domain idn there is only one option and the option name may therefore be omitted. The name of the field is defined as an &aux field. The value of the field must be a Common Lisp string. In the production that defines domain expr there is also an &aux definition. It defines an auxiliary field for every option of that domain.

The AST compiler compiles the abstract syntax definition into a set of creator and accessor functions and type predicates. These functions check that the types of their arguments are legal.

Manipulating abstract syntax trees only with the functions that the AST compiler generates would be tedious in some situations. In all structure oriented functions, e.g., functions that walk through syntax trees, we must find out the type of the node and then branch according to the type. When a new node type is implemented all branching expressions must be changed so that they reflect the new situation.

Methods provide a solution to this problem. A *generic function* is defined and then, for every node type, a method is written that implements the generic function. The generic function is called whenever we must branch according to the type of the node. When a new node type is defined we only have to write methods for that node type. Thus, the effects of defining (or deleting or changing) a node type are local.

Methods may be inherited. The abstract syntax definition implies a three-level class hierarchy. The domains may be considered as abstract superclasses and the options as instantiable subclasses of their domain class. All domains of a syntax have a single abstract superclass, namely the whole syntax. Because the option classes are the only instantiable classes every node belongs to three classes: one option class, one domain class, and one syntax class.

A method can be defined for the whole syntax, for a whole domain, or for a single option. The methods that are defined for the whole syntax may be redefined for a domain or for an option, and the methods that are defined for a domain may be redefined for some options of that domain.

There are situations where this method facility is not very useful. For example, using methods to implement a program that copies abstract syntax trees is not very much easier than implementing it with normal functions. We must write copy method for every node type. Although all copy methods are rather similar we cannot use inheritance effectively. Maintaining these methods after changes to the abstract syntax definition is also tedious.

To solve these problems *structural methods* were included in AST. A structural method is a macro that expands to a set of methods, one for each node type. In a structural method definition the programmer defines abstractly how nodes should be processed according to their structure.

5.1.3 DG

DG [50] (Descriptor Generator) is a general purpose object class definition language that contains multiple inheritance and methods.

As opposed to most object oriented languages, a DG definition defines a set of classes, not just a single class. This permits an easy implementation, because all classes can be compiled at the same time. However, the compilations after modifications last longer.

Here is an example of a DG definition:

```
(define-descriptors windows
   :structures
   ((window
     :fields ((name string)
              (top int)
              (bottom int)
              (left int)
              (right int)
              (superior (&opt window))
              (inferiors (&list window))))
```

```
(process
 :fields ((top-level-function (&union symbol lambda))))
(bordered-window-mixin
 :instantiable nil
 :fields ((borders (&union int (&list int) symbol))))
(bordered-window-process
 :parents (window process bordered-window-mixin))
(:extern int :test fixnump)
(:extern string :test stringp)
(:extern symbol :test symbolp)
(:extern lambda :test (lambda (x)
                        (and (listp x)
                             (eq (first x) 'lambda)))))
```

The name `windows` identifies the definition. It is used as a prefix in the creation of operation and class names. The list after keyword `:structures` contains the class definitions. In the example eight classes are defined, namely `window`, `process`, `bordered-window-mixin`, `bordered-window-process`, `int`, `string`, `symbol`, and `lambda`. A class definition consists of a set of field definitions and an optional parent class definition. A parent class definition contains one or more class names, thus multiple inheritance is permitted. A field consists of the name and the type of the field. The type is either a class or a type expression built of class names and type expression operations. Type expression operations are: `&opt` meaning an optional field, `&list` meaning a list field, and `&union` meaning a disjoint union. For example, the value of the field `inferiors` of an object of the class `window` is a list of objects belonging to the class `window` or to some of its descendant classes. Type expressions may be nested. For example, the type of the field `borders` defines that the value of this field must be either an integer (defining the same border thickness for all borders), a list of integers (defining different thicknesses for different borders), or a symbol (a name of a variable that should be bound to either an integer or to a list of integers when an instance is created).

An object of a class has the fields that are defined in the definition of that class and all the fields that are defined in the ancestors of the class. Fields cannot be redefined in subclasses.

Because DG is not a self contained language there must be a facility for using objects of other kinds as field values of DG objects. This is accomplished with the `:extern` class definition. It contains the name of the external class and optionally a type predicate. External classes may be used in field type definitions in the same way as normal DG classes. However, an external class may not be the parent of some other class. In our example the classes `int`, `string`, `symbol`, and `lambda` are external.

Some classes represent behavior that is only reasonable when that class is combined with some other class. In our example, the class `bordered-window-mixin` is such a class. When we combine this class with some window class we can make windows that have border lines of variable width. It is not reasonable to have objects of class `bordered-window-mixin` and therefore the class definition contains the definition `:instantiable nil` which prohibits instantiation.

The DG compiler compiles the class definitions to a set of creator and accessor functions and type predicates.

Methods and structural methods may be written for DG classes in the same way as in AST.

DG contains also a facility for defining *access oriented fields*. These fields have functions associated with them that are called whenever the field is accessed. This mechanism is used in the implementation of the compilation control strategy of the XE compiler.

5.1.4 Experiences

The ROCC, AST and DG languages helped the development of the XE compiler in many ways.

The XE syntax was modified several times during the development of the language. Because of ROCC, these modifications were rather easy to implement. However, the XE syntax was rather complicated and it was not easy to describe it as an LALR(1) syntax. ROCC was also very slow (the compilation of the XE syntax takes about one hour in the Symbolics Lisp Machine). Probably the parser implementation would have been easier if we had an LL parser generator that produces a set of recursive parsing functions and a mechanism for hand coding those parts of the parser that cannot be described in the LL form.

The AST and DG definitions are compact. In the case of AST this is achieved because AST is a special purpose language. In the case of DG this is mainly achieved with inheritance. The method definition facilities localize changes to the abstract syntax and class definitions, because the code that uses objects only calls generic functions. This kind of polymorphism is very beneficial and one might wonder whether this should also have been included in the XE language.

Maybe the biggest savings in programming and maintenance work were achieved with structural methods. The structural method definitions are compact. The XE compiler contains 10 structural method definitions. The size of a definition varies from 10 lines to 94 lines. The total size of the structural method definitions is 349 lines. The size of the macro expansion results is 18,800 lines, thus on the average the expansion ratio is 54. Hand coding would have somewhat reduced this figure, but anyway the savings have been thousands of lines. We do not know any other languages that contain structural methods, but this kind of facility would be very useful in other languages too.

The connection between the AST and DG languages and other data structure definitions, e.g. `defstructs`, that we used in the XE compiler, was a big problem. The `&aux` fields of the AST definitions and the `:extern` classes in the DG definitions did not solve the problem well. The problem would have been avoided if we had implemented AST on top of DG which would have been rather easy. We should also not have used any other data structure definition facilities. If we had had one uniform data structure definition language it would have been possible to implement a disk save facility for the XE environment. In the current situation it turned out to be a too complicated a task and could not be performed efficiently.

5.2 The XE compiler

The structure of the XE compiler is shown in figure 2. In addition to the modules that can be found in any typical compiler the XE compiler contains a couple of special modules that are needed either because of the special features of the XE language, e.g., the parameterization, or because of the goals that we had in mind when designing the XE programming environment, e.g., the avoidance of unnecessary recompilations. In the next sections, we describe briefly the parts of the XE compiler. A more detailed description of can be found in [7].

5.2.1 Data structures and symbol table

Two forms of syntax trees are used in the XE compiler. The parser builds a parse tree called *PAST*. The checker traverses PAST and builds an abstract syntax tree called *CAST*. CAST is annotated with context information that is needed in context condition checking and code generation.

The reason for the use of two different syntax trees is the complex syntax of XE. The same syntactic structures are used for denoting semantically different structures. For example, an identifier followed by a list of expressions closed in brackets can be either a generator instantiation or a syntactic sugar form of a `fetch` or a `store` operation. The use of two forms of syntax trees keeps the parser reasonably simple, because in order to build PAST the parser does not have to be fully aware of the context in which it is parsing the input. However, this choice made the checker more complicated. The lesson learned is that the language syntax should be reasonably clear in order to avoid such parsing problems.

PAST and CAST are defined with the *AST* abstract syntax definition tool. PAST consists of 39 productions and 114 options and CAST consists of 11 productions and 73 options. The reason for the smaller amount of productions and options in CAST than in PAST is that CAST is annotated with context information, e.g., types.

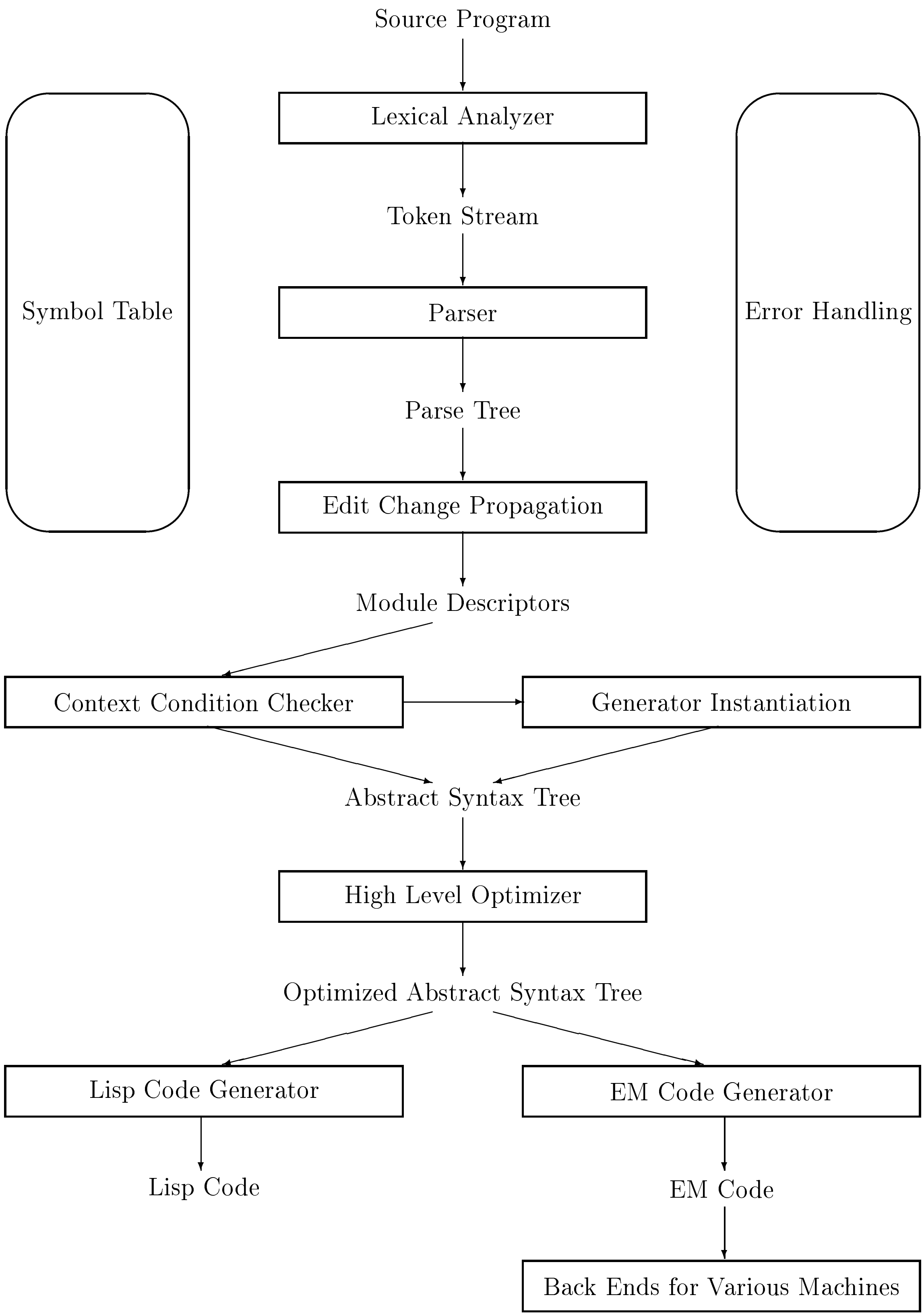

Figure 2: The Structure of the XE compiler.

The types and other context information is stored in *descriptors*. Descriptors are defined using the *DG* language. The XE descriptor definition contains 41 classes, 10 of which are not instantiable, i.e., they are abstract superclasses used only in the definition of other classes. There are descriptors for procedures, iterators, rules, and data types, descriptors for procedure, iterator, rule, and data type generators and generator instances, descriptors for arguments, local and own variables, descriptors for signals and signal handlers, and so on. The descriptors refer to other descriptors and other data structures such as CAST trees.

The descriptors are stored in a symbol table. XE allows the same name to be used for referring to entities of different kinds at the same time. For example, there can be a signal `foo` and a procedure `foo` defined at the same time. Some definitions can also be shadowed by new definitions of the same name. The generation mechanism of the XE programming environment (see [39]) makes things even more complicated. For these reasons, descriptors are searched using not only their name, but also the scope in which the reference is found, the type of the entity that is searched for, and the generation in which the compilation takes place. The XE symbol table is implemented as a hash table.

5.2.2 Error handling

The error handling facility consists of data structures for representing error messages, Common Lisp functions and macros for signaling errors, and 144 predefined error message strings.

The error message data structure contains the error message string and information about the location in which the error was found. The XE programming environment contains various facilities for scanning error messages. For instance, it is possible to go through error messages in an editor buffer so that the cursor is positioned at the point in the source text where the error was found.

5.2.3 Lexical analyzer

The lexical analyzer scans the input stream and returns tokens. In addition to the token values, the tokens also contain information of the position where the token was found in the source stream. This information is used, for example, with error messages.

5.2.4 Parser

The XE parser is written in ROCC. It consists of 89 productions with 234 different right-hand sides. The input to the parser is a stream of tokens that the lexical analyzer produces. The actions in the productions construct PAST nodes using the results of the subforms of the production. The parser also does some very simple context checks.

5.2.5 Edit change propagation

The edit change propagation module implements the smart recompilation algorithm of the XE compiler. It detects the set of XE modules that should be compiled after a part of a program is edited. As input, the edit change propagation module receives two lists of parse trees. Typically, the first list contains the parse trees that were found in editor buffers before the previous compilation and the second list contains the parse trees that are currently found in the buffers. The output is a list of module descriptors that should be recompiled. The smart recompilation algorithm is described in the next chapter.

5.2.6 Context condition checker

The context condition checker is the most complicated module of the XE compiler. The checker has several tasks. It checks that the context conditions are satisfied, e.g., that the referred identifiers are defined and that the definitions match with the use. Because XE is a quite complicated a language there are vast amounts of different checks that have to be performed. The checker also has to find out what exact syntactic form it is checking, because the parser does not distinguish all syntactic structures from each other, as was pointed out above. Finally, the checker builds CAST trees and descriptors that contain information that the optimizers and the code generators use.

5.2.7 Generator instantiation

A special module is needed for the instantiation of the generators. In the XE programming environment instantiation, i.e., the substitution of formal parameters with the actual parameters is performed at the PAST level. Thus, when a generator is instantiated the PAST representation of the instance is built from the PAST representation of the generator. The information stored in the descriptors of the generator is not used to determine the structure of the instance. The resulting PAST is processed by the checker as a normal non-parameterized module to form the descriptors and CAST representation needed.

The choice of making instantiations at the PAST level simplified the compiler. On the other hand, it also slows down the XE compilation very much. Typically, an XE program instantiates lots of predefined generators, and the instances refer to instances of other generators. Thus, the compilation of a short, well chosen, piece of XE code can take several minutes.

5.2.8 High-level optimizer

The high-level optimizer modifies CAST trees. It applies in-line coding and tail recursion removal to both procedures and iterators. Applying these optimizations yields code that contains simple `while` loops instead of recursion or `for` or `iterate` statements. These optimizations are crucial in a language that is based on abstract data types, because the operations of abstract data types are usually very small and without in-line coding considerable procedure and iterator call overheads would result.

A second kind of high-level optimization is the removal of unnecessary procedures and iterators. The instantiation of data type generators results to many routines that are never used in the program. The high-level optimizer is capable of detecting and removing such procedures. It is also able to combine similar operations, for example operations that access records in the similar way. Without such optimizations the target system would often grow too big.

5.2.9 Code generation

The XE environment contains two code generators, one for the program development environment and one for the embedded target environment.

In the programming environment, the XE programs are compiled into Lisp code. The Lisp code generator does a straightforward transformation from the CAST trees to Common Lisp code. The Lisp code is then compiled using the Symbolics in-core Lisp compiler.

The Intel code generation has been built upon the tools of the ACK (Amsterdam Compiler Kit) [67]. The Intel code generation phase produces ACK virtual machine code called EM. The EM code has to be moved into a Unix machine having the ACK installed. It is further compiled by ACK Intel code generator to an Intel assembler file, which can be compiled with Intel assembler to a 8086 object code. The result is then loaded into the embedded environment.

5.2.10 Compilation order

The XE environment supports separate compilation of modules. Nested submodules, e.g., routines of a data type or local procedures are also compiled separately.

A set of modules cannot be compiled (or recompiled) in arbitrary order. The information that is needed in the compilation of a module must be computed before the module is compiled. The compilation order can be determined from dependencies between modules. The usual rule for determining the order is that if module A depends on module 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 [34] 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 module.

A similar method can be used when compiling a set of modules for the first time. In this case the modules 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 module and when a reference to another module, that is not already compiled, is detected the compilation of the first module is suspended and the compilation of the second module is started. When that compilation has completed the compilation of the first module is resumed. In order to avoid deadlocks in case of recursive dependencies, all parts of the second module are not compiled. Only those parts that are needed in the compilation of the first module are compiled. It should be noted that no separate algorithm is needed for solving cyclic dependencies as in [34]. According to our experience the average number of suspended compilations is small.

6 Smart recompilation

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.

The problem of smart recompilation consists of the detection of the program units that must be recompiled after a change to some unit and the detection of compilation order for those units. Smart recompilation is also called incremental compilation. The incrementally compiled units may vary from statements to program modules or files.

In this section, the recompilation algorithm of the XE compiler is described and compared with other recompilation algorithms. Although the algorithm was designed for the recompilation of XE programs it can be used in the recompilation of other languages too.

6.1 Background

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

The detection of the set of modules that must be compiled is tedious and therefore algorithms have been implemented for recompilation. The MAKE program [21] 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 analyzed in order to see what modules within a file have changed and which other modules see the changes. The dependency relation is too coarse and causes unnecessary recompilations. For example, adding a new comment to some module in a file causes the recompilation of that file and all files that contain modules that depend on some modules of this file.

6.2 The XE recompilation algorithm

The XE recompilation algorithm determines the set of modules that are affected when some modules are changed. The algorithm is phase oriented, which means that for all affected modules it is known what parts of the compiler need be run in recompilation. No compilation order need be explicitly determined for the affected modules because the demand driven compilation method (described in section 5.2.10) is used also in recompilations. Here we present a slightly simplified version of the algorithm. The original algorithm uses a few more dependency classes and change propagation rules than the algorithm presented here, but the basic idea is the same.

The algorithm is based on compilation dependencies that the checker and other phases of the compiler detect. A *header dependency* is established from module A to module 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 in-line coding) detects *in-line dependencies*. If the call of routine B is in-line coded in A then an in-line dependency is established from A to B. It should be noted that an in-line dependency never exists alone, but is always combined with a body dependency.

The detection of the affected modules consists of two phases. In the first phase the edit changes are analyzed and a *change set* is computed in a similar way as in Tichy's algorithm [68]. In the second phase the changes are propagated using the compilation dependencies and the set of affected modules 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 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, *body* if only the body was changed, and *equates* if only the equates inside the module have changed (in this case the equates inside the module are processed in the similar manner). Note that the edit change analysis filters out all changes that do not change the syntactic structure of the program. For example, changing the indentation of the program or modifying comments does not cause any recompilations.

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

$M_{recompile}$ consists of three parts. M_{header} contains all modules that must be recompiled because the static semantics of their header may have changed. M_{body} contains all modules that must be recompiled because the static semantics of their body (but not the header) may have changed. $M_{in-line}$ contains all modules whose static semantics has not changed but for which code must be generated again because of in-line dependencies. The sets are computed using the following rules.

Rule1 If the parse tree of module M is in $T_{modified}$ and the change type is *header* then M is in M_{header}.

Rule2 If the parse tree of module M is in $T_{modified}$ and the change type is *body* then M is in M_{body}.

Rule3 If there is a header dependency from module M_1 to module M_2 and M_2 is either in M_{header} or in $M_{deleted}$ then M_1 is in M_{header}.

Rule4 If there is a body dependency from module M_1 to module M_2 and M_2 is either in M_{header} or in $M_{deleted}$ then M_1 is in M_{body}.

Rule5 If there is an in-line dependency from module M_1 to module M_2 and M_2 is either in M_{body} or in $M_{in-line}$ then M_1 is in $M_{in-line}$.

Rule6 If there is a header dependency from module M_1 to module M_2 and there is a module M_3 in M_{new} and M_3 shadows M_2 in M_1 then M_1 is in M_{header}.

Rule7 If there is a body dependency from module M_1 to module M_2 and there is a module M_3 in M_{new} and M_3 shadows M_2 in M_1 then M_1 is in M_{body}.

After change propagation the modules in M_{new} and $M_{recompile}$ are compiled. Only the code generator has to be applied on modules in $M_{in-line}$. For other modules the checker is first called and after that, if no errors were detected, the code generator is applied. Only the body of the modules in M_{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 modules in $M_{recompile}$ and gives the user an opportunity to edit the modules.

The algorithm cannot guarantee that the set $M_{recompile}$ is minimal, i.e., that it contains only the modules that are really affected. However, the algorithm filters out unnecessary recompilations quite well. There are only a few of cases in which the static semantics are not actually changed, but the algorithm causes recompilations. These cases do not occur very often. Also it would probably be more expensive to detect these cases than to recompile.

6.3 An example

As an example, we present a small program that consists of a couple of simple modules. Note that the program has no sensible meaning and it is not an example of good XE programming style. The only purpose of the example is to demonstrate how the recompilation algorithm works.

```
a = datatype is create

    rep = d

    create = proc(i: d) returns(a)
    end
end

b = proc(i, j: d) returns(a)
    return(a$create(i + j))
end

c = iter(i: d) yields(a)
    for x: d in [1 .. i] do
      yield(b(i, x))
    end
end

d = int
```

The data type `a` contains only one operation, `create` that creates new objects of type `a` when given an object of the representation type. The representation type of `a` is `d`. The procedure `b` creates an object of type `a` by combining two objects of type `d` with the '+' operation. The iterator `c` iterates from 1 to `i` yielding the result of an application of `b` with arguments `i` and `x`. The type constant `d` is equated to `int`.

Let us now look how the recompilation algorithm works. $T_{current}$ contains `a`, `b`, `c`, and `d`. T_{old} is empty. The result of the edit change analysis is the following. T_{new} contains `a`, `b`, `c`, and `d`. $T_{deleted}$ and $T_{modified}$ are empty. After the change propagation M_{new} contains new modules corresponding `a`, `b`, `c`, `d`, `a$create`, and `a$rep`. $M_{deleted}$ and $M_{recompile}$ are empty. Thus, modules `a`, `b`, `c`, `d`, `a$create`, and `a$rep` will be checked and code will be generated for them.

After the compilation, the following dependencies exist between the modules:

- The data type `a` refers to the procedure `a$create` (body reference).
- The data type `a` refers to the constant `a$rep` (body reference).
- The procedure `a$create` refers to the data type `a` (header reference).
- The procedure `a$create` refers to the constant `d` (header reference).
- The constant `a$rep` refers to the constant `d` (header reference).
- The procedure `b` refers to the data type `a` (header reference).

- The procedure `b` refers to the constant `d` (header reference).
- The procedure `b` refers to the procedure `a$create` (body reference).
- The procedure `b` refers to the procedure `int$add` (body reference).
- The iterator `c` refers to the data type `a` (header reference).
- The iterator `c` refers to the constant `d` (header reference).
- The iterator `c` refers to the iterator `int$from_to_by` (body reference).
- The iterator `c` refers to the procedure `b` (body reference).
- The constant `d` refers to the data type `int` (header reference).

Let us now assume that the procedure `a$create` is modified in the following way.

```
a = datatype is create

    rep = d

    create = proc(i: d) returns(a)
        return(up(i))
    end
end
```

The result of the edit change analysis is the following: $T_{modified}$ contains `a` (the type of the change is *equates*) and `a$create` (the type of the change is *body*). T_{new} and $T_{deleted}$ are empty. Using rule 2 of the change propagation we insert the module `a$create` into M_{body}. Because no other rules are applicable the result is that `a$create` should be recompiled.

Let us now suppose that the value of `d` is changed to `longint`.

```
d = longint
```

After the edit change analysis $T_{modified}$ contains `d` (the type of the change is *header*). T_{new} and $T_{deleted}$ are empty. Using rule 1 with $M = $ `d` we insert `d` into M_{header}. Using rule 3 with $M_1 = $ `b` and $M_2 = $ `d` we insert `b` into M_{header}. Similarly we insert `c`, `a$create`, and `a$rep` into M_{header}. Using rule 4 with $M_1 = $ `a` and $M_2 = $ `a$rep` we insert `a` into M_{body}. Some other rules are also applicable, but they do not change the result which is: M_{header} contains `b`, `c`, `d`, `a$create`, and `a$rep`, and M_{body} contains `a`. Thus all modules should be recompiled.

Let us now assume that a third argument is inserted into the argument list of the procedure `b`.

```
b = proc(i, j, k: d) returns(a)
    return(a$create(i + j + k))
end
```

The result of the recompilation algorithm is that `b` and `c` should be recompiled. When `c` is checked, an error is detected, because the call of `b` in the body of `c` contains only two actual arguments. After the correction of the error, only `c` has to be recompiled.

```
c = iter(i: d) yields(a)
    for x: d in [1 .. i] do
      yield(b(i, x, 1))
    end
end
```

6.4 Other recompilation algorithms

Dausmann describes various recompilation schemes using different levels of change analysis [18]. 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 describes the recompilation algorithm used in the implementation of the Mary2 language [55]. The algorithm first recompiles the changed module. If the export interface of the module changes, then each module 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 ([55]). In [36] it is shown 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 modules from the actual recompilation. Also the grain size of recompilation is different.

Tichy presents a smart recompilation algorithm [68] 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 modules. Both algorithms also compute transitive closures of references (or dependencies), but the closures are used for different purposes. Tichy's algorithm uses program files as compilation units whereas our algorithm uses procedures, iterators, etc. as compilation units. Thus the program development turn-around time cannot be reduced so much using Tichy's algorithm as ours.

Scwanke and Kaiser ([60]) have improved Tichy's algorithm by permitting benign type inconsistencies between separately compiled modules. This enchanced algorithm has the same problem as the original one: too big compilation units.

Hood, Kennedy and Müller present 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 [34]. The algorithm uses the compilation dependency graph that we described in section 5.2.10. The nodes of the graph are visited in topological order to detect the affected modules, and data-flow 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 data-flow analysis of dependencies. Both algorithms can handle recursive dependencies.

Olsson and Whitehead ([52]) present `srm` — a system that automatically generates a Makefile that reflects the dependencies among the modules of a program. They also describe a 'semi-smart' algorithm that does not propagate unnecessary dependencies in the generated Makefile. Also this algorithm uses bigger compilation units than our algorithm.

The potential benefits of smart recompilation and related techniques have also been studied. Linton and Quong [42] describe an empirical study on how much time programmers spend waiting for compiling and linking, how many modules are compiled each time a program is linked, and the change in size of the compiled modules. The study was performed by instrumenting the MAKE program so that statistics can be collected. The results indicate that usually a program is remade by compiling a small number of modules (program files) no matter how large a program is, and that these modules rarely change by more than 100 bytes. The implication is that an incremental compilation and linking environment is feasible because most changes are already incremental in nature. The results show also that the time for compilation consistently dominates the time for linking [42]. According to these results, smart recompilation algorithms, such as the one that is used in the XE environment, will considerably reduce the program development "turnaround time".

7 Conclusions

In this chapter we summarize the main issues presented, discuss the results that we have achieved and describe some directions of further research.

7.1 Summary of the main issues

The ExBed project was founded to determine how expert system techniques can be applied within embedded systems running on microprocessors. Because rule-based programming is the most widely used implementation technique in knowledge-based software development it was taken as the basis of the research. Literature lead to the choice of the forward chaining inference strategy. It was decided to approach the problem, not by using or designing an expert system shell, but by embedding the required facilities in a general purpose programming language. The resulting environment was to consist of two distinct parts: the development environment, where embedded applications are programmed and tested, and the actual embedded run-time environment.

Several problems were found in using traditional rule-based languages for embedded applications. These problems were related to the RETE pattern matching algorithm that is used in many of these languages, the data representation and program structuring facilities provided by the languages, the interaction with procedural software, and the software reliability (see section 2.2).

Our solution to these problems was twofold. Firstly, we abandoned the RETE algorithm. Instead, we used non state-saving pattern matching algorithms. Secondly, we combined rule-based and procedural programming in the same language by using abstract data types as the unifying component (see section 2.3).

In order to test our ideas we implemented three programming languages: XC, XD, and XE.

XC is a rule-based extension of C++. The basic constructs of XC are working memories, rules, and rulesets. A working memory contains a collection of objects that the rules access. A rule consist of a condition and an action. A ruleset is similar to a C++ class. In addition to data and functions, it contains a set of rules. The XC compiler is a preprocessor that compiles the XC program into a C++ program. It creates a forward chaining inference engine for every ruleset. The embedding of XC programs is based on the services provided by the underlying operating system; the language itself does not contain any constructs for this purpose. XC was used in the implementation of a couple of applications, and it proved to be a useful tool. The data abstraction approach was found to be a suitable basis for the implementation of embedded expert systems. However, the XC programming environment that was based on the C++ environment was a disappointment. Also, yet increased openness of the language was desired (see chapter 3).

In the second phase of the project a more ambitious approach was taken. Instead of extending an existing language we decided to design and implement a new general purpose programming language XE. XE combines abstract data types, procedures, iterators, and rule-based programming. Parameterization is used as the type generalization mechanism. In order to support rule-based programming, XE contains the rule construct. A rule is a special kind of iterator that yields rule instantiations. XE does not contain working memories and rule sets as language constructs. Instead, these constructs — as well as the inference engine and the conflict resolution — are easily programmed using the other constructs of XE and can be provided in a library. Also the embedding of XE programs is mainly based on the use of the services provided by the underlying operating systems. XE contains, however, a couple of constructs for this purpose. XE was used in the implementation of XEDA, a digital switch diagnostic advisor application. XD is the implementation of the XE rule abstraction on top of C and C++ (see chapter 4).

The XE compiler is not an independent program. Its parts were integrated into a uniform programming environment, together with facilities for program editing and version management. The environment was implemented in Common Lisp on top of the Symbolics Genera operating system. Various metaprogramming tools were used in the implementation (see chapter 5).

In the development of the XE programming environment one of the major goals was to minimize the amount of recompilation in response to changes in source modules. The XE recompilation algorithm determines the set of modules that have to be recompiled after a change to some module. The algorithm is phase oriented, i.e., for all affected modules it is known what parts of the compiler need be run in recompilation. The detection of modules that have to be recompiled consists of two parts. First, the edit changes are analyzed and a change set is computed. Then the changes are propagated using compilation dependencies that have been detected between modules (see chapter 6).

7.2 Discussion

Combining rule-based and procedural programming in the same language by using abstract data types as the unifying component proved to be a fruitful approach. This conclusion was made in every application where our languages — XC, XD, and XE — were used. Using the same language for implementing both the conventional and the knowledge-based parts of an application made the task easier for the programmer. There was no need to memorize different lexical and syntactical forms. What is more important, there was no need to declare same things twice — in the rule-based language and in the procedural language — and then to maintain these two sets of declarations. Abstract data types proved to enforce program clarity and correctness, as expected. They provided a uniform data representation facility that could be used both in the procedural and in the rule-based code.

The choice of abandoning the RETE algorithm and using non state-saving algorithms instead is more controversial. In our measurements, the non state-saving pattern matching algorithm that was used in XC proved to be more efficient (both in respect of time and space) than the RETE algorithm used in OPS83. However, it is easy to provide examples where the RETE (or the TREAT) algorithm is more efficient. More practical experiments are required before it can be stated which kind of applications are more common. It should be noted that this is a more technical issue than the choice of using abstract data types as the data representation facility. It may also be independent of the choice of using abstract data types, but this should be shown by implementing a language that combines abstract data types and RETE pattern matching.

In the design of XC and XE, different approaches were used. XC was an extension of an existing language whereas XE was a fully new language. The benefits of designing a new language instead of extending an existing one are clear. The new ideas can be expressed more easily in a new language and the resulting language tends to be more uniform. On the other hand, the task of designing a new general purpose programming language is much more difficult than the task of extending an existing language. Also a big problem remains: it is hard to get users to a new language.

The implementations of XC and XE were very different. The implementation of XC was based on preprocessing while XE was implemented by writing a compiler from scratch and integrating it with other program development tools. The implementation of XC took a couple of weeks whereas the implementation of XE took a couple of man-years.

At the time when the decision was made, to design and implement XE instead of further developing XC, good tools for the development of C++ programs were not available. The C++ compilers were based on precompilation and they contained many errors. There were no debuggers. At that time the XE approach seemed to be the only way to get a good programming environment for the development of embedded expert systems. The current situation is different. The C and C++ compilers and debuggers and various other tools of the GNU-project [64, 69, 63, 62] are now available and can be used in the development of extensions of C and C++. XD, an extension of C and C++ that contains the XE rule mechanism, was easily implemented using these facilities.

7.3 Directions of further research

More comparisons between the RETE, the TREAT, and the non state-saving pattern matching algorithms in the context of embedded expert systems are needed.

The improvements to the RETE algorithm that have been proposed lately should be examined. Also the combination of RETE and TREAT with abstract data types should be studied. The user definable optimizations that can be used with non state-saving algorithms should be more thoroughly examined. The possibilities of combining the benefits of the different pattern matching algorithms in a single system should be studied, either by designing a new adaptive algorithm or by providing the user a mechanism for selecting a pattern matching algorithm from among a set of predefined algorithms.

A programming environment where these experiments can be performed should be designed and implemented. It would probably be easiest to implement it on top of the GNU environment and use C++ as the basis for the programming language. The possibilities of including a smart recompilation facility to this environment should also be studied.

References

[1] A. V. Aho and S. C. Johnson. LR Parsing. *ACM Computing Surveys*, 6(2):99–124, June 1974.

[2] J. Arkko. Asiantuntijajärjestelmäkielen optimoitu toteutus Intel-ympäristöön (Optimized Implementation of an Expert System Language). Master's thesis, Helsinki University of Technology, Laboratory of Information Processing Science, 1989. In Finnish.

[3] J. Arkko, N. Bouteldja, V. Hirvisalo, J. Kuusela, E. Nuutila, and M. Tamminen. Overview of the XE Language Environment. Technical Report TKO-C41, Helsinki University of Technology, Laboratory of Information Processing Science, Espoo, Finland, 1989.

[4] J. Arkko, V. Hirvisalo, J. Kuusela, E. Nuutila, and M. Tamminen. Notes on XE Run Time Characteristics and Code Generation. Technical Report TKO-C37, Helsinki University of Technology, Laboratory of Information Processing Science, Espoo, Finland, 1989.

[5] J. Arkko, V. Hirvisalo, J. Kuusela, E. Nuutila, and M. Tamminen. Programming in XE. Technical Report TKO-C33, Helsinki University of Technology, Laboratory of Information Processing Science, Espoo, Finland, 1989.

[6] J. Arkko, V. Hirvisalo, J. Kuusela, E. Nuutila, and M. Tamminen. ROCC Programmer's Guide. Technical Report TKO-C38, Helsinki University of Technology, Laboratory of Information Processing Science, Espoo, Finland, 1989.

[7] J. Arkko, V. Hirvisalo, J. Kuusela, E. Nuutila, and M. Tamminen. XE Programming Environment Implementation Notes. Technical Report TKO-C40, Helsinki University of Technology, Laboratory of Information Processing Science, Espoo, Finland, 1989.

[8] J. Arkko, V. Hirvisalo, J. Kuusela, E. Nuutila, and M. Tamminen. XE Reference Manual. Technical Report TKO-A26, Helsinki University of Technology, Laboratory of Information Processing Science, Espoo, Finland, 1989.

[9] J. Arkko, V. Hirvisalo, J. Kuusela, Esko Nuutila, and M. Tamminen. Some Experiences with Rules in Procedural Programming Languages. In *International Computer Science Conference '88*, pages 712–718, Hong Kong, December 1988. IEEE Computer Society.

[10] J. Arkko, V. Hirvisalo, J. Kuusela, Esko Nuutila, and M. Tamminen. Rule-Based Expression Mechanisms for Procedural Languages. *Computational Intelligence*, 5(4), 1989.

[11] J. Arkko, J. Kuusela, E. Nuutila, and M. Tamminen. Filex: A File System Expert Written in XC. In *Proceedings of the Finnish AI Symposium*, pages 544–552, Helsinki, Aug. 15-18 1988. Limes Ry.

[12] J. Arkko, J. Kuusela, E. Nuutila, M. Tamminen, and V. Hirvisalo. The ExBed project — some experiences. In *Proceedings of the Finnish AI Symposium*, pages 496–505, Helsinki, Aug. 15-18 1988. Limes Ry.

[13] F. Barachini and N. Theuretzbacher. The Challenge of Real-time Process Control for Production Systems. In *Proceedings of the AAAI-88 Seventh National Conference on Artificial Intelligence, Volume 2*, pages 705–709, Los Altos, CA, 1988. Morgan Kaufmann Publishers, Inc.

[14] N. Bouteldja. Engineering Knowledge-Based Software for Large Embedded Systems. Helsinki University of Technology, Laboratory of Information Processing Science, 1989. Tech. Lic. Thesis.

[15] N. Bouteldja, J. Arkko, V. Hirvisalo, J. Kuusela, E. Nuutila, and M. Tamminen. Building an Embedded Knowledge-Based Application using Abstractions. In *Proceedings of the 2nd Scandinavian Conference on Artificial Intelligence*, pages 735–748, Tampere, Finland, 1989.

[16] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5*. Addison-Wesley, Reading, Massachusetts, 1985.

[17] B. D. Clayton. ART Programming Tutorial. Technical report, Inference Corporation, Los Angeles, CA, March 15 1985.

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

[19] L. Fagan. *Ventilator manager: a program to provide on line consultative advice in the intensive care unit*. PhD thesis, Stanford University, Computer Science Department, 1980.

[20] F. Fages. On the proceduralization of rules in expert systems. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 119–138. Elsevier Science Publishers B.V., 1988.

[21] S. Feldman. Make — A Program for Maintaining Computer Programs. *Software Practice and Experience*, 9(3):255–265, March 1979.

[22] C. L. Forgy. *On the Efficient Implementation of Production Systems*. PhD thesis, Carnegie-Mellon University, Department of Computer Science, 1979.

[23] C. L. Forgy. OPS5 User's Manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, Department of Computer Science, 1981.

[24] C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[25] C. L. Forgy. The OPS83 Report. Technical Report CMU-CS-84-133, Carnegie-Mellon University, Department of Computer Science, 1984.

[26] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[27] K. Gorlen. Object-Oriented Program Support, OOPS Version 1. Technical report, National Institutes of Health, 1986.

[28] J.H. Griesmer, S.J. Hong, M. Karnaugh, M.I. Schor, R.L. Ennis, D.A. Klein, K.R. Milliken, and H.M. VanWoerkom. A continuous real time expert system. In *Proceedings of the AAAI'84 National Conference*, pages 130–136, 1984.

[29] A. Gupta, C. Forgy, A. Newell, and R. Wedig. Parallel Algorithms and Architectures for Rule-Based Systems. *ACM SIGARCH Computer Architecture News*, 14(2):28–37, June 1986.

[30] A. Gupta and M. Tambe. Suitability of Message Passing Computers for Implementing Production Systems. In *Proceedings of the AAAI-88 Seventh National Conference on Artificial Intelligence, Volume 2*, pages 687–692, Los Altos, CA, 1988. Morgan Kaufmann Publishers, Inc.

[31] F. Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9), September 1985.

[32] V. Hirvisalo, J. Arkko, J. Kuusela, E. Nuutila, and M. Tamminen. XE Design Rationale: CLU Revisited. *ACM SIGPLAN Notices*, 24(9):72–79, September 1989.

[33] V. Hirvisalo, J. Arkko, J. Kuusela, E. Nuutila, and M. Tamminen. XE Utility Library Manual. Technical Report TKO-C34, Helsinki University of Technology, Laboratory of Information Processing Science, Espoo, Finland, 1989.

[34] R. Hood, Kennedy K., and H. A. Müller. Efficient Recompilation of Module Interfaces in a Software Development Environment. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 180–189, Palo Alto, California, Dec. 9-11 1986.

[35] T. Ishida. Optimizing Rules in Production System Programs. In *Proceedings of the AAAI-88 Seventh National Conference on Artificial Intelligence, Volume 2*, pages 699–704, Los Altos, CA, 1988. Morgan Kaufmann Publishers, Inc.

[36] H-M. Järvinen. Principles of Intelligent Recompilation. Private communication, 1988.

[37] F. C. Karonis and R. E. King. Multi-Level Expert Systems for Industrial Control. In *Proceedings of the International Expert Systems Conference*, Oxford, June 2-4 1987. Learned Information.

[38] B. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[39] J. Kuusela. An Integrated Programming Environment for the XE Language. Helsinki University of Technology, Laboratory of Information Processing Science, 1990. Tech. Lic. Thesis.

[40] J. Kuusela, J. Arkko, V. Hirvisalo, E. Nuutila, and M. Tamminen. XE Programming Environment User's Guide. Technical Report TKO-C35, Helsinki University of Technology, Laboratory of Information Processing Science, Espoo, Finland, 1989.

[41] J. Kuusela and E. Nuutila. Hybrid AI Development Tools. In *Proceedings of the Finnish AI Symposium, vol. 2*, pages 149–156, Espoo, Aug. 19-22 1986. Otapaino.

[42] M. A. Linton and R. W. Quong. A Macroscopic Profile of Program Compilation and Linking. *IEEE Transaction on Software Engineering*, 16(4):427–436, April 1989.

[43] B. Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *The CLU Reference Manual.* Springer-Verlag, Berlin, 1981. Lecture Notes in Computer Science 114.

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

[45] J. McDermott, A. Newell, and J. Moore. The Efficiency of Certain Production System Implementations. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern-Directed Inference Systems*, pages 155–176. Academic Press, Inc., Orlando, Florida, 1978.

[46] K. R. Milliken, A. V. Cruise, R. L. Ennis, J. L. Hellerstein, M. J. Masullo, M. Rosenbloom, and H. M. van Woerkom. YES/L1: A Language for Implementing Real-Time Expert Systems. Technical Report RC-11500, IBM Thomas J. Watson Research Center, Yorktown Height, New Your, 1986.

[47] D. P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proceedings of the AAAI-87 Sixth National Conference on Artificial Intelligence, Volume 1*, pages 42–47, Los Altos, CA, 1987. Morgan Kaufmann Publishers, Inc.

[48] P. Nayak, A. Gupta, and P. Rosenbloom. Comparison of the Rete and Treat Production Matchers for Soar (A Summary). In *Proceedings of the AAAI-88 Seventh National Conference on Artificial Intelligence, Volume 2*, pages 693–698, Los Altos, CA, 1988. Morgan Kaufmann Publishers, Inc.

[49] E. Nuutila. Asiantuntijajärjestelmien rakentamistyökalun sääntöpohjainen päättelymekanismi (Rule-based deduction mechanism for an expert system building tool). Master's thesis, Helsinki University of Technology, Laboratory of Information Processing Science, 1985. In Finnish.

[50] E. Nuutila, J. Arkko, V. Hirvisalo, J. Kuusela, and M. Tamminen. AST and DG Tools Programmer's Guide. Technical Report TKO-C39, Helsinki University of Technology, Laboratory of Information Processing Science, Espoo, Finland, 1989.

[51] E. Nuutila, J. Kuusela, M. Tamminen, J. Veilahti, J. Arkko, and N. Bouteldja. XC — A Language for Embedded Rule Based Systems. *ACM SIGPLAN Notices*, 22(9):23–32, September 1987.

[52] R. A. Olsson and G. R. Whitehead. A simple technique for automatic recompilation in modular programming languages. *Software Practice and Experience*, 19(8), August 1989.

[53] D. L. Parnas. On the criteria to be used in composing systems into modules. *Communications of the ACM*, 15(12), December 1972.

[54] G. A. Pascoe. Elements of Object-oriented programming. *BYTE*, August 1986.

[55] M. Rain. Avoiding Trickle-Down Recompilation in the Mary2 Implementation. *Software Practice and Experience*, 14(12):1149–1157, December 1984.

[56] J. Rintanen, M. Tamminen, J. Arkko, V. Hirvisalo, J. Kuusela, and E. Nuutila. XD Manual. Technical Report TKO-C36, Helsinki University of Technology, Laboratory of Information Processing Science, Espoo, Finland, 1989.

[57] J. Robertson. 'STIMULUS' — a base language for real time expert systems. In *Proceedings of the Conference on AI and advanced Computer Technology*, Wiesbaden, September 22-24 1985.

[58] P.A. Sachs, A.M. Paterson, and M.H.M. Turner. Escort - an expert system for complex operations in real time. *Expert Systems*, 3(1):22–29, 1986.

[59] Marshall I. Schor, Timothy P. Daly, Ho Soo Lee, and Beth R. Tibbitts. Advances in RETE pattern matching. In *Proceedings of the AAAI-86 Fifth National Conference on Artificial Intelligence, Volume 1, Science*, pages 226–232, Los Altos, CA, 1986. Morgan Kaufmann Publishers, Inc.

[60] R. W. Schwanke and G. E. Kaiser. Smarter Recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4), October 1988.

[61] D. E. Shaw. NON-VONS's Applicability to Three AI Task Areas. In *International Joint Conference on Artificial Intelligence*, 1985.

[62] R. M. Stallman. *GNU Emacs Manual.* Free Software Foundation, Inc., Cambridge, Massachusetts, October 1986.

[63] R. M. Stallman. *GDB Manual. The GNU Source-Level Debugger*. Free Software Foundation, Inc., Cambridge, Massachusetts, November 1989.

[64] R. M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Inc., Cambridge, Massachusetts, January 1990.

[65] Salvatore J. Stolfo, Daniel Miranker, and David E. Shaw. Architecture and Applications of DADO: A Large-Scale Parallel Computer for Artificial Intelligence. In *International Joint Conference on Artificial Intelligence*, 1983.

[66] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[67] A. S. Tanenbaum, H. Van Staveren, E. G. Keizer, and J. W. Stevenson. A Practical Toolkit for Making Portable Compilers. *Communications of the ACM*, 26(9):654–660, September 1983.

[68] W. F. Tichy. Smart Recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.

[69] M. D. Tiemann. *User's Guide to GNU C++*. Free Software Foundation, Inc., Cambridge, Massachusetts, August 1989.

[70] J. H. Walker, D. A. Moon, D. L. Weinreb, and M. MacMahon. The Symbolics Genera Programming Environment. *IEEE Software*, November 1987.

[71] M. L. Wright, M. W. Green, G. Fiegl, and P. F. Cross. An expert system for real-time control. *IEEE Software*, 3(2):16–24, March 1986.

[72] K-E. Årzen. Expert systems for process control. In *Proceedings of the first International Conference on Applications of AI in Engineering Practice*, Southampton, UK, April 15-18 1986.