Tool Support for Language Extensibility

Jan Bosch
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: Jan.Bosch@ide.hk-r.se
www: http://www.pt.hk-r.se/~bosch

Abstract
During the last years, one can recognise a development towards application domain languages and extensible language models. Due to their extended expressiveness, these language models have considerable advantages over rigid general purpose languages. However, a complicating factor in the use of extensible language models are the conventional compiler construction techniques. Compilers constructed using these techniques often are large entities that are highly complex, difficult to maintain and hard to reuse. As we have experienced, these characteristics clearly complicate extending existing compilers. As a solution to this, we developed an alternative approach to compiler construction is proposed, based on object-oriented principles. The approach is based on delegating compiler objects (DCOs) that provide a structural decomposition of compilers in addition to the conventional functional decomposition. The DCO approach supports modularisation and reuse of compiler specifications, such as lexer and parser specifications. We constructed an integrated tool set, LETOS, implementing the functionality of delegating compiler objects.

1 Introduction

Although the way software systems are constructed is changing constantly, many underlying principles have remained constant. One example of such a principle is the use of a single general purpose programming language for programming the complete software system. Software engineers considered it most productive to work within the context of a single language model for all subsystems, independent of the particular characteristics of the subsystems. Independent of the domain for which the software system was constructed, the same programming language is used.

Lately, one can recognise a development in which the use of a general purpose language is no longer necessarily considered to be the optimal solution. The use of application domain languages is increasing in domains where general purpose languages clearly lack expressiveness, such as graphical user interfaces and robot languages. The application domain language is designed such that the main concepts in the application domain have equivalent concepts in the programming language.

Each application domain, or even science domains, has an associated paradigm that is used by the experts working in that domain. A paradigm [Kuhn 62] can be defined as a set of related concepts with underlying semantics [Bosch 95c]. For the domain experts, the paradigm provides the ‘language’ to talk about phenomena in the domain. The concepts represent relevant abstractions in the domain. A paradigm is not a static entity, but a dynamic complex of related concepts that is changing constantly, both in the number of concepts but also in the semantics of existing concepts.

When constructing executable specifications of applications in an application domain, the software engineer has to convert the concepts of the application domain into the concepts (or constructs) supported by the programming language. The difficulty of the translation process is depending on the ‘conceptual distance’ between the domain and programming language concepts, i.e. the *semantic gap*. An important lesson learned by the software engineering community is that minimising the semantic gap is highly beneficial and improves the understandability and maintainability of software. Application domain languages are a

logical consequence of this conclusion, since these languages aim at a one-to-one relation between the concepts in the application domain and the programming language, but also other approaches exist.

When one tries to decrease the semantic gap between the domain and programming language, two fundamental approaches are available, the revolutionary and the evolutionary approach. The revolutionary approach discards the concepts in the general purpose programming language and starts from scratch, basing the language design solely on the concepts in the application domain. The evolutionary approach starts with a general purpose language model that is extended with application domain specific concepts. Thus, the language model is extensible with constructs that, among others, may represent application domain concepts. In [Bosch 95c], we introduced the notion of *paradigm extensibility* to refer to this principle.

Both approaches have both advantages and disadvantages. An important advantage of the evolutionary approach is that software developed using the extensible language model can relatively easy be integrated with other software developed using the same basic language model, but with different extensions. Since virtually all software systems cover multiple application domains, integration is an important property. A disadvantage, however, is that extensions of the language model are required to uniformly extend the semantics of the existing language constructs. This limits the possible extensions of the language model.

The problem addressed in this paper concerns both the revolutionary and the evolutionary approach. The traditional compiler construction techniques provide no or little support for language extensibility or efficient construction of application domain languages. The existing techniques primarily aim at providing support for the construction of large and rigid general purpose languages. No support is provided for dealing with the complexity of compiler development, extensibility of compiler specifications and reusability of existing compiler specifications. As a solution to these problems, we defined the concept of delegating compiler objects (DCOs) [Bosch 95b]. Based on this theoretical concept, we developed an integrated tool set, LETOS, supporting compiler construction based on DCOs. Using the tool set, two compilers for our extensible object model, i.e. the layered object model (**LayOM**), were developed. These compilers compile **LayOM** code into C and C++ code, respectively.

The remainder of this paper is organised as follows. In the next section, the problems that we identified while constructing compilers for extensible languages are discussed. In section 3, the theoretical concept of delegating compiler objects is described. Section 4 is concerned with the integrated tool set, LETOS. In section 5 our extensible object model **LayOM** is discussed and its DCO-based compiler to C++ is described. The paper is concluded in section 7.

2 Problems of Compiler Construction

Traditionally, a compiler is constructed using a number of components that are invoked in a chronological manner. A typical compiler consists of a *lexer*, a *parser*, a *semantic analyser* and a *code generator*. The compilation process is decomposed towards the different *functions* that convert program code into a description in another language. From our experiences in the construction of extensible language models, we have identified four problems of this approach to compiler construction:

- **Complexity**: A traditional, monolithic compiler tries to deal with the complexity of a compiler application through decomposing the compilation process into a number of subsequent phases. Although this indeed decreases the complexity, this approach is not *scaleable* because a large problem cannot *recursively* be decomposed into smaller components. In order to deal with compilers that are changed and extended on a regular basis, the one level decomposition into lexing, parsing, semantic analysis and code generation phases we have experienced to be insufficient.

- **Maintainability**: Although the compilation process is decomposed into multiple phases, each phase itself can be a large and complex entity with many interdependencies. Maintaining the parser, for example, can be a difficult task when the syntax description is large and has many interdependencies between the production rules. In the traditional approaches, the syntax description of the language cannot be decomposed into smaller, independent components.
- **Reuseability and extensibility**: Although the domain of compilers has a rich theoretical base, building a compiler often means starting from scratch, even when similar compiler specifications are available. The notion of reusability has no supporting mechanism in compiler construction. In addition, no support is available for extending an existing compiler with new expressiveness.
- **Tool support**: Since we aim at constructing extensible and maintainable compilers, we have experienced that the tools implementing the traditional compiler construction techniques are not really supportive. Especially the lack of modularisation of parser and lexer specifications and the batch-oriented approach of the tools is problematic.

Concluding, we can state that the conventional approach to compiler construction suffers from a number of problems when applied to the construction of compilers for application domain languages and extensible language models. In the remainder of the paper, we discuss our solution to the identified problems and the tool support that we developed.

3 Delegating Compiler Objects

The main goal of the delegating compiler object (DCO) approach [Bosch 95b] is to achieve modular, extensible and maintainable implementations of compilers. In section 2, it was concluded that the existing approaches to compiler construction do not provide the features required for application domain languages and extensible language models. Several problems related to the *complexity* of compiler development, *extensibility* of compiler components, *reusability* of elements of an existing compiler and the lack of tool support were identified.

As an alternative for the conventional, monolithic approach, we proposed *delegating compiler objects* (DCOs), a novel concept for compiler development. The philosophy of DCOs is that next to the functional decomposition into a lexer, parser and code generator, another decomposition dimension is offered, i.e. structural decomposition. The structural decomposition is used for the primary decomposition. Rather than having a single compiler consisting of a lexer, parser and code generator, an input text can be compiled by a group of compiler objects that cooperate to achieve their task. A compiler object, when detecting that a particular part of the syntax is to be compiled, can instantiate a new compiler object and delegate the compilation of that particular part to the new compiler object.

Each compiler object consists of one or more lexers, one or more parsers and a parse graph. The parser, during parsing, constructs a *parse graph* consisting of parse graph nodes and connections between these nodes. These nodes contain the code generation knowledge and when the compiler object receives a request for generating the output code, this request is delegated to the parse graph. The nodes in the parse graph generate output code.

The delegating compiler object concept is based on the concepts of *parser delegation* and *lexer delegation* for achieving the structural decomposition of grammar, respectively, lexer specification. These techniques will be described later in the paper.

3.1 Delegating Compiler Objects

A delegating compiler object (DCO) is a generalisation of a conventional compiler in that the conventional compiler is used as a component in the DCO approach. In our approach, a compiler object consists of one or

more lexers, one or more parsers and a parse graph. The parse graph consists of parse graph node objects described in section.

The underlying assumption when defining delegating compiler objects is that a programming language can be decomposed into a set of major concepts in that language. For each of these concepts a compiler object can be defined. Each compiler object contains information on how to instantiate and interact with other compiler objects. The compilation process starts with the instantiation of an initial compiler object. This compiler object can instantiate other compiler objects and delegate parts of the compilation to these delegated compiler objects. These compiler objects can, in turn, instantiate other compiler objects and delegate the compilation to them. The result of a compilation is a set of compiler objects that can be accessed through the base compiler object.

A traditional object model, consisting of a number of main language constructs, such as *class*, *method*, *object* and *inheritance*, when implemented as a DCO-based compiler could have an architecture as shown in figure 1. Each DCO has a lexer, parser and parse graph for the particular object model component. The parser of the base DCO, i.e. class, instantiates the other DCOs and delegates control over parts of the compilation process to the instantiated DCOs.

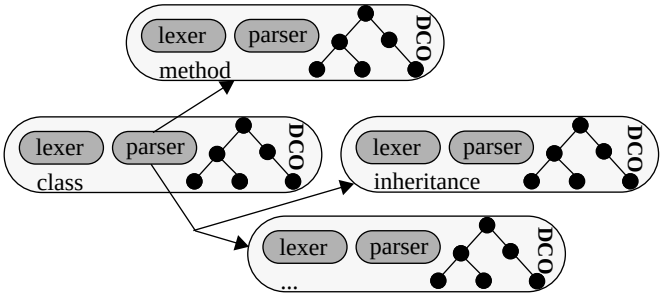

Figure 1. Example DCO-based compiler

The concept of delegating compiler objects makes use of *parser delegation* [Bosch 95a], *lexer delegation* and parse graph node objects [Bosch 95c]. These techniques will be discussed in the following sections.

3.2 Parser Delegation

Parser delegation is a mechanism that allows one to modularise and to reuse grammar specifications. In case of modularisation, a parser can instantiate other parsers and redirect the input token stream to the instantiated parser. The instantiated parser will parse the input token stream until it reaches the end of its syntax specification. It will subsequently return to the instantiating parser, which will continue to parse from the point where the subparser stopped. In case of reuse, the designer can specify for a new grammar specification the names of one or more existing grammar specifications. The new grammar is extended with the production rules and the semantic actions of the reused grammar(s), but has the possibility to override and extend reused production rules and actions.

We define a monolithic grammar as G=(I, N, T, P), where I is the name of the grammar, N is the set of nonterminals, T is the set of terminals and P is the set of production rules. The set $V = N \cup T$ is the vocabulary of the grammar. Each production rule $p \in P$ is defined as p=(q, A), where q is defined as $q = x \rightarrow \alpha$ where $x \in N$ and $\alpha \in V^*$ and A is the set of semantic actions associated with the production rule q.

Parser delegation extends the monolithic grammar specification in several ways to achieve reuse and modularisation. First, while defining a new grammar, one can specify grammars that are to be reused by the new grammar. When partially equivalent grammar specifications exists, one would like to reuse an existing grammar and extend and redefine parts of it. If a grammar is reused, all the production rules and semantic actions become available to the reusing grammar specification. Parser delegation implements reuse of an

existing grammar by creating an instance of a parser for the *reused* grammar upon the instantiation of the parser for the *reusing* grammar. The reusing parser uses the reused parser by *delegating* parts of the parsing process to the reused parser.

When modularising a grammar specification, the grammar specification is divided into a collection of grammar module classes. When a parser object decides to delegate parsing, it creates a new parser object. The active parser object delegates parsing to the new parser object, which will gain control over the input token stream. The new parser object, now referred to as the *delegated* parser, parses the input token stream until it is finished and subsequently it returns control to the delegating parser object.

In addition to delegating to a different parser, the parser can also delegate control to a new compiler object. In that case, rather than a new parser, new DCO is instantiated and the active DCO leaves control to the new DCO. The delegated DCO compiles its part of the input syntax. When it is finished it returns control to the delegating compiler object.

To describe the required behaviour, the production rule of a monolithic parser has been replaced with a set of production rule types. These production rule types control the reuse of production rules from reused grammars and the delegation to parser and compiler objects. Parser delegation employs the following production rule types:

- n : v_1 v_2 ... v_m, where $n \in N$ and $v_i \in V$

  All productions $n \rightarrow V^*$ from G_{reused} are excluded from the grammar specification and only the productions n from $G_{reusing}$ are included. This is the *overriding* production rule type since it overrides all productions n from the reused grammars.

- n +: v_1 v_2 ... v_m, where $n \in N$ and $v_i \in V$

  The production rule n : v_1 v_2 ... v_m, if existing in G_{reused} is replaced by the specified production rule n. The *extending* production rule type facilitates the definition of new alternative right hand sides for a production n.

- n [id]: v_1 v_2 ... v_m, where $n \in N$ and $v_i \in V$

  The element id must contain the name of a parser class which will be instantiated and parsing will be delegated to this new parser. When the delegated parser is finished parsing, it returns control to the delegating parser. The results of the delegated parser are stored in the parse graph. When $i is used as an identifier, v_i must have a valid parser class name as its value. The *delegating* production rule type initiates delegation to another parser object.

- n [[id]]: v_1 v_2 ... v_m, where $n \in N$ and $v_i \in V$,

  The element id must contain the name of a delegating compiler object type which will be instantiated and the process of compilation will be delegated to this new compiler object. When the delegated compiler object is finished compiling its part of the program, it returns the control over the compilation process to the originating compiler object. The originating compiler object receives, as a result, a reference to the delegated compiler object which contains the resulting parse graph. The delegating parser stores the reference to the delegated DCO in the parse graph using a DCO-node. Next to using an explicit name for the id, one can also use $i as an identifier, in which case v_i must have a valid compiler object class name as its value. The DCO production rule type causes the delegation of the compilation process to another DCO.

In figure 2, the process of parser delegation for modularising purposes is illustrated. In (1) a delegating production rule is executed. This results (2) in the instantiation of a new, dedicated parser object. In (3) the control over the input token stream has been delegated to the new parser, which parses its section of the token stream. In (4) the new parser has finished parsing and it has returned the control to the originating

parser. This parser stores a reference to the dedicated parser as it contains the parsing results. Note that the lexer and parse graph are not shown for space reasons.

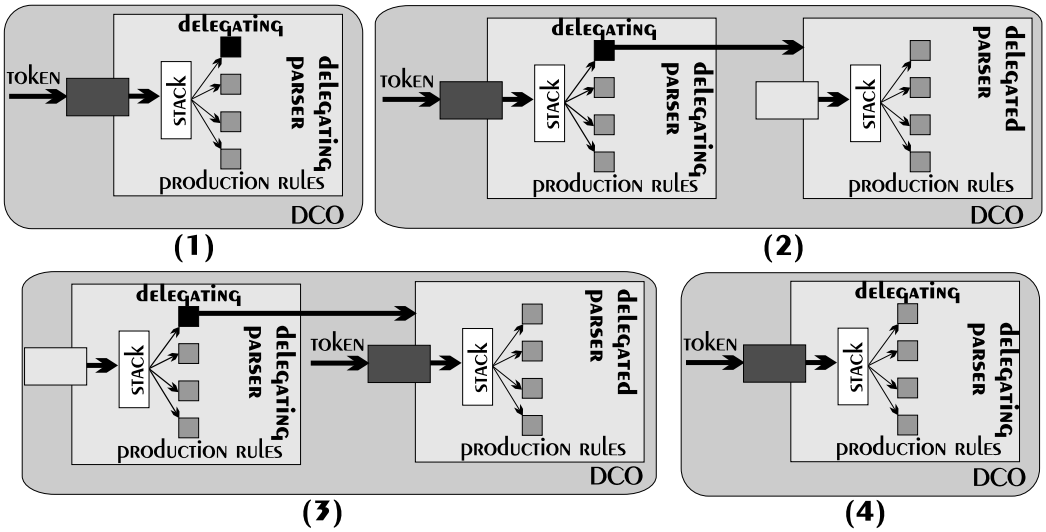

Figure 2. Parser Delegation for Grammar Modularisation

We refer to [Bosch 95a, Bosch95c] for more detailed discussion of parser delegation.

3.3 Lexer Delegation

The *lexer delegation* concept provides support for *modularisation* and *reuse* of lexical analysis specifications. Especially in domains where applications change regularly and new applications are often defined modularisation and reuse are very important features. Lexer delegation can be seen as an *object-oriented* approach to lexical analysis.

A monolithic lexer can be defined as L = (I, D, R, S), where I is the identifier of the lexer specification, D is the set of definitions, R is the set of rules and S is the set of programmer subroutines. Each definition $d \in D$ is defined as d=(n, t), where n is a name, $n \in N$, the set of all identifiers, and t is a translation, $t \in T$, the set of all translations. Each rule $r \in R$ is defined as r = (p, a), where p is a regular expression, $p \in P$, the set of all regular expressions, and a is an action, $a \in A$, the set of all actions. Each subroutine $s \in S$ is a routine in the output language which will be incorporated in the lexer generated by the lexer generator. Different from most lexical analysis specifications languages, a lexer specification in our definition has a *identifier* which will be used in later sections to refer to different lexical specifications.

Lexer delegation, analogous to parser delegation, extends the monolithic lexer specification to achieve modularisation and reuse. The designer, when defining a new lexer specification, can specify the lexer specifications that should be reused by the new lexer. When a lexer specification is reused, all definitions, rules and subroutines from the reused lexer specification become available at the reusing lexer specification. In a lexical specification, the designer is able to exclude or override *definitions*, *rules* and *subroutines*. Overriding a reused definition d = (n, t) is simply done by providing a definition for d in the reusing lexer definition. One can, however, also *extend* the translation t for d by adding a =+ behind the name n of d. Extending a definition is represented as $n =+ t_{extended}$.The result of extending this definition is $n = t_{extended} ? t_{reused}$.

A reused rule r = (p, a) can also be overridden by defining a rule r' = (p, a'), i.e. a rule with the same regular expression p. One can interpret extending a rule in two ways. The first way is to interpret it as extending the action associated with the rule. The second way is to extend the regular expression associated with an

action. Both types of rule extensions are supported by *lexer delegation*. Extending the regular expression p is represented as p' |+ p.

When a lexer specification is modularised, it is decomposed into smaller modules that contain parts of the lexer specification. One of the modules is the *initial lexer* which is instantiated at the start of the lexing process. The extensions for lexer modularisation consist of two new actions that can be used in the action part of rules in the lexer specifications. Lexer delegation occurs in the action part of the lexing rules. The semantics of these actions are the following:

- **Delegate(<lexer-class>)**: This action is part of the action part of a lexing rule and is generally followed by a *return(<token>)* statement. The delegate action instantiates a new lexer object of class *<lexer-class>* and installs the lexer object such that any following token requests are delegated to the new lexer object. The delegate action is now finished and the next action in the action block is executed.
- **Undelegate**: The undelegate action is also contained in the action part of a lexing rule. The undelegate action, as the name implies, does the opposite of the delegate action. It changes the delegating lexer object such that the next token request is handled by the delegating lexer object and delegation is terminated. The lexer object does not contain any state that needs to be stored for future reference, so the object is simply removed after finishing the action block.

For a more detailed discussion of lexer delegation we refer to [Bosch 95c].

3.4 Parse Graph Nodes

In the delegating compiler object approach, an object-oriented, rather than a functional approach, is taken to parse tree and code generation. Instead of using passive data structures as the nodes in the parse tree as was done in the conventional approach, the DCO approach uses *objects* as nodes. A node object is instantiated by a production rule of the parser. Upon instantiation, the node object also receives a number of arguments which it uses to initialise itself. Another difference from traditional approaches is that, rather than having an separate code generation function using the parse tree as data, the node objects themselves contains knowledge for generating the output code associated with their semantics.

A parse graph node object, or simply node object, contains three parts of functionality. The first is the *constructor* method, which instantiates and initialises a new instance of the node object class. The constructor method is used by the production rules of the parser to create new nodes in the parse graph. The second part is the *code generation* method, which is invoked during the generation of output code. The third part consists of a set of methods that are used to access the state of the node object, e.g. the name of an identifier or a reference to another node object.

The grammar has facilities for parse graph node instantiation. An example production rule could be the following:

```
method   :  name ‘(' arguments ‘)' ‘begin' temps statements ‘end'
            [ MethodNode($1, $3, $6, $7) ]
         ;
```

The parse graph, generally, consists of a large number of node objects. There is a root object that represents the point of access to the parse graph. When the compiler decides to generate code from the parse graph, it sends a `generateCode` message to the root node object. The root node object will generate some code and subsequently invoke its children parse nodes with a `generateCode` message. The children parse nodes will generate their code and invoke all their children.

4 LETOS

While developing the Language Extensibility TOol Set (LETOS), we had two goals. The main goal, obviously, was to develop a tool that implemented the concept of delegating compiler objects. The second goal, however, was *effectiveness*, i.e. achieving the tool functionality against minimal effort. Therefore, we based ourselves explicitly on existing tools, such as YACC and LEX, and extended these tools with the lacking functionality.

Figure 3 shows the user interface of LETOS. The user can open and work with one project (i.e. compiler) at a time. The user interface of the tool consists of three lists. The left upper list contains the a list of the DCOs that are part of the compiler. The user of the tool can add, rename and delete DCOs from the list. In addition, the user can specify which DCO should be the initial DCO. The initial DCO is instantiated upon instantiation of the compiler. In the next section, the compiler constructed for **LayOM** is discussed that generates C++ output code. This compiler consists of several DCOs that, depending on the input source text, all might be instantiated and used. The *project* button contains an option that will generate an executable compiler based on the specification of the DCOs, the grammar specifications, the lexer specifications and the parse graph node classes.

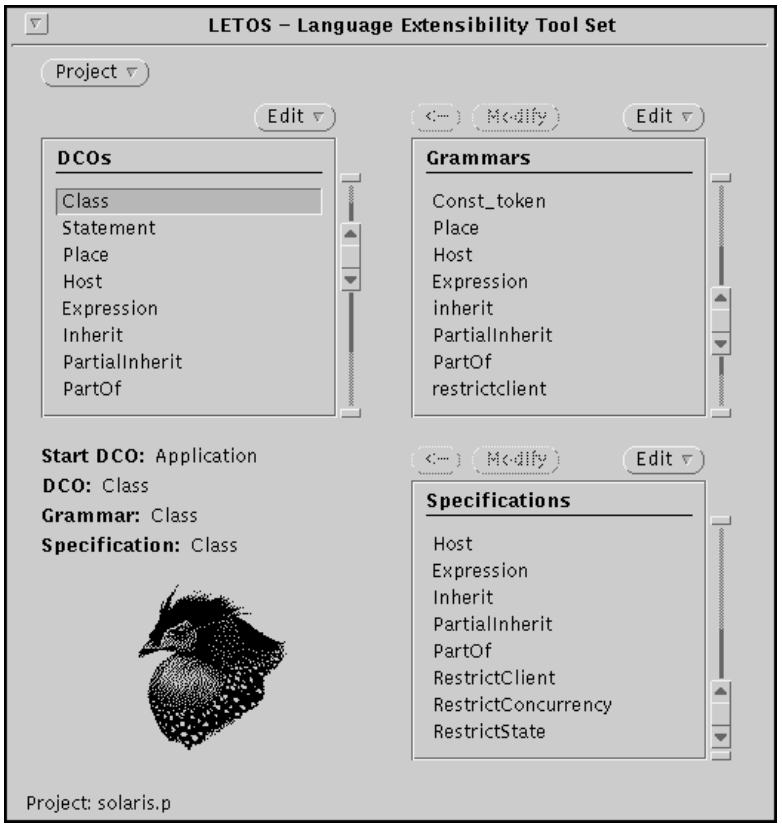

Figure 3. Overview of LETOS

The right upper list in figure 3 contains the various grammar specifications that are part of one of the DCOs. The user can define new grammars, add existing grammars to the project, delete grammars from the project and modify the grammar specification. The arrow button is used to associate the currently selected grammar with the currently selected DCO. In this way, DCOs can be configured with other grammars very easy. The right lower list in figure 3 presents the lexer specifications. The lexer specifications are also a part of the DCOs.

When generating a compiler, each grammar and lexer specification is translated into a corresponding C++ file. Subsequently, a makefile is generated and the C++ compiler is invoked. For pragmatic reasons, LETOS makes use of YACC for converting the grammar specification into a corresponding C++ file. By doing this

we were not required to build a parser generator, but could focus our effort on constructing a translator and a C++ preprocessor. The grammar translation can be seen as composed of three steps. Each grammar and all grammars it reuses are converted into one YACC grammar specification. The YACC specification is converted into a C++ program by YACC. A preprocessor will convert the resulting C++ program into an equivalent C++ program in which all standard YACC identifiers are renamed to unique names. The lexer specifications are treated in an analogous manner. Each DCO is translated into a C++ class that provides an interface to the lexing and parsing functions and the parse graph. Based on this, each DCO and all associated functionality is merely a C++ class from the outside and can be instantiated as such.

In the next section, our extensible object model is described, i.e. **LayOM**. We have constructed two compilers converting **LayOM** code into C++ code. The compilers are based on delegating compiler objects and have been built using LETOS.

5 Example: Layered Object Model

As an example of the use of the techniques and tools for language extensibility, we introduce our research language, the layered object model (**LayOM**) in section 5.1. Part of the implementation of a compiler for **LayOM** to C++ are discussed in section 5.2. The section is concluded in section 5.3.

5.1 Layered Object Model

The layered object model is an extensible object model, but currently a **LayOM** object contains, next to the traditional object model components as class, method and instance variable, a number of additional components such as *layers*, *states* and *categories*. In figure 4, an example **LayOM** object is presented. The layers encapsulate the object, so that messages send to or by the object have to pass the layers. Each layer, when it intercepts a message, converts the message into a passive message object and evaluates the contents to determine the appropriate course of action. Layers can be used for various types of functionality. Layer classes have been defined for the representation of relations between objects, but also for representing design patterns and object communication patterns.

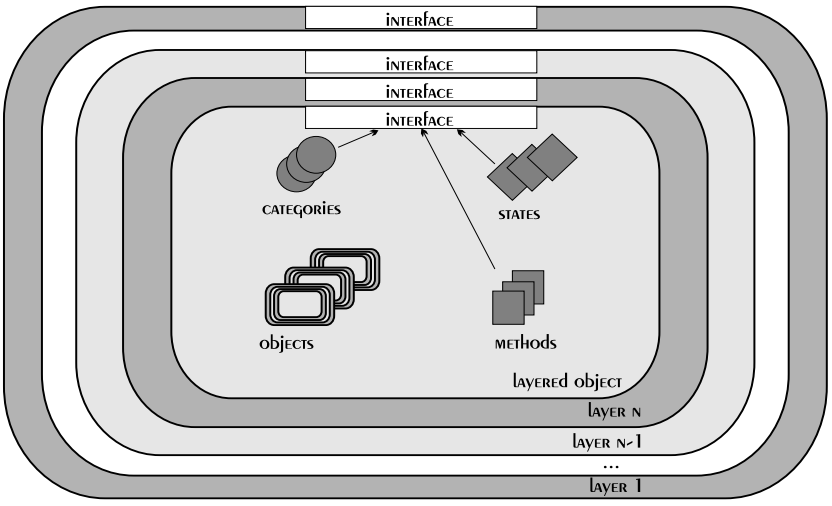

Figure 4. The layered object model

A *state* in **LayOM** is an abstraction of the internal state of the object. In **LayOM**, the internal state of an object is referred to as the *concrete state*. Based on the object's concrete state, the software engineer can define an externally visible abstraction of the concrete state, referred to as the *abstract state* of an object. The abstract object state is generally simpler in both the number of dimensions, as well as in the domains of the state dimensions.

A category is an expression that defines a client category. A client category describes the discriminating characteristics of a subset of the possible clients that should be treated equally by the class. The behavioural layer types use categories to determine whether the sender of a message is a member of a client category. If the sender is a member, the message is subject to the semantics of the specification of the behavioural layer type instance.

A layer, as mentioned, encapsulates the object and intercepts messages. It can perform all kinds of behaviour, either in response to a message or otherwise. Layers have, among others, been used to represent relations between objects. In **LayOM**, relations have been classified into structural relations, behavioural relations and application-domain relations. *Structural relation* types define the structure of a class and provide *reuse*. These relation types can be used to *extend* the functionality of a class. Inheritance and delegation are examples of structural relation types. The second type of relations are the *behavioural relations* that are used to relate an object to its clients. The functionality of the class is used by client objects and the class can define a behavioural relation with each client (or client category). Behavioural relations *restrict* the behaviour of the class. For instance, some methods might be restricted to certain clients or in specific situations. The third type of relations are *application domain relations*. Many domains have, next to reusable application domain classes, also application domain relation types that can be reused. For instance, the *controls* relation type is a very important type of relation in the domain of process control. In the following two sections, structural and behavioural relation layer types will be discussed.

As mentioned earlier, the layered object model is an *extensible* object model, i.e. the object model can be extended by the software engineer with new components. **LayOM** can, for example, be extended with new layer types, but also with new object model components, such as *events*. One could say that the notion of extensibility, which is a core feature of the object-oriented paradigm, has been applied to the object model itself. Object model extensibility may seem useful in theory, but in order to apply it in practice it requires extensibility of the translator or compiler associated with the language. In the case of **LayOM**, classes and applications are translated into C++ or C. The generated classes can be combined with existing, hand-written C++ code to form an executable. Since the **LayOM** compiler is based on *delegating compiler objects*, we discuss the implementation of the **LayOM** compiler in the next section as an example of the use of DCOs and LETOS.

5.2 LayOM Compiler

In the system that is currently under development, the layered object model compiler is implemented as a set of delegating compiler object that are instantiated during parsing and compile parts of the input syntax. This means that delegating compiler object classes have been defined for *Class*, *Method*, *State*, *Category*, *Application* and all layer types. In figure 5, the structure of the delegating compiler objects (DCOs) for the *class* part of the layered object model compiler is shown. The figure only shows the delegation to separate DCOs; the reuse connections are not shown.

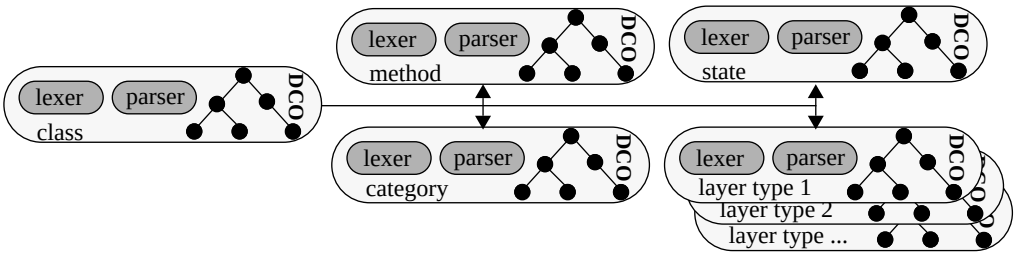

Figure 5. LayOM Class Compiler

The layered object model compiler takes as an input either a class or an application expressed in **LayOM** syntax. A **LayOM** class is translated into a C++ class specification and a **LayOM** application is translated into a C++ main function. The **LayOM** implementation environment makes use of the C++ compiler and several

libraries to translate a **LayOM** application that has been converted into a C++ main function into an executable file.

The **LayOM** environment is intended to execute on Sun workstations running Solaris. This operating system supports threads and light-weight processes which are to be used by the **LayOM** environment to achieve concurrency. The execution environment uses several additional Solaris features.

5.2.1 Class DCO

The `Class` compiler contains the knowledge of scanning and parsing a **LayOM** class and generating the C++ code for the class. In addition, it can instantiate compilers for components that are part of the class specification, e.g. `methods` or `states`.

Below, the syntax rules of the `Class` parser are shown. The `Class` parser is more of a shell parser that instantiates compiler objects for all components except for the instance variables of the class. The `Class` parser generates a compiler object for each state, category, method and layer. For a layer, a compiler object indicated by the layer type is instantiated.

```
Class[ObjectDecl]

class                 : CLASS ID body ';'
                      ;
body                  : layers states categories methods variables
                      ;
layers                : LAYERS layerbody
                      | LAYERS /* empty */
                      ;
layerbody             : layerbody
                      | layerbody lstatement
                      ;
lstatement [[$3]]     : ID ':' ID
                      ;
states                : STATES statebody
                      | STATES /* empty */
                      ;
statebody             : sstatement
                      | statebody sstatement
                      ;
sstatement [[State]]  : ID
                      ;
categories            : CATEGORIES categorybody
                      | CATEGORIES /* empty */
                      ;
categorybody          : cstatement
                      | categorybody cstatement
                      ;
cstatement [[Category]]: ID
                      ;
variables             : INSTVAR objectdecls
                      | INSTVAR /* empty */
                      ;
methods               : METHODS methodbody
                      | METHODS /* empty */
                      ;
methodbody            : mstatement
                      | methodbody mstatement
                      ;
mstatement [[Method]] : ID
                      ;
```

The `Class` parser only contains the integrating functionality related to a class definition. One of its primarily tasks is to instantiate the DCOs for the layers, methods, categories and states. Only the nested objects are parsed by the class DCO itself.

5.2.2 State DCO

The `State` DCO compiles an individual state definition. The `State` parser, shown below, reuses the syntax definitions of the `expression` grammar specification and defines the declaration syntax for a state. The size of the syntax specification has been reduced to a single production rule due to the effective reuse of the `Expression` syntax.

```
State[Expression]

state          :  RETURNS ID BEGINN  expression END ';'
```

The first line declares a grammar with the identifier `State` which reuses from another grammar called `Expression`. Since a state only specifies an expression that maps the internal concrete state of an object to a new domain and all grammar specifications for expressions are reused, the grammar specification is very small, i.e. a single production rule.

5.3 Conclusion

The use of the delegating compiler object approach and its supporting tool set, LETOS, has been proven to be very beneficial for the development of the **LayOM** compilers. In the previous section, we have shown how the **LayOM** compiler has been decomposed into a set of DCOs and we have shown how parts of the DCO specification can reuse existing grammar or lexer specifications. Due to the space limitations, we are unable to describe more aspects of the compiler. Instead, we refer to [Bosch 95c, Pheasant 95] for more detailed information.

6 Related Work

In [Dearle 88], a persistent system for compiler construction is proposed. The approach is to define a compiler as a collection of modules with various functionality that can be combined in several ways to form a compiler family. The modules have a type description which is used to determine whether components can be combined. The approach proposed in [Dearle 88] is different from the DCO approach in the following aspects. First, although the approach enhances the traditional compiler modularisation, modularisation and reuse of individual modules, e.g. the grammar specification, is not supported. Secondly, judging from the paper, it does not seem feasible to have multiple compilers cooperating on a single input specification, as in the DCO approach.

In [Järnvall et al. 95] a different approach to language engineering, TaLE, is presented. Rather than using a meta-language like Lex or YACC for specifying a language, the user edits the classes that make up the implementation using a specialised editor. TaLE not immediately intended for the implementation of traditional programming languages, but primarily for the implementation of languages that have more dynamic characteristics, like application-oriented languages. The TaLE approach is different from our approach in, at least, two aspects. First, TaLE does not make use of metalanguages like LEX and YACC, whereas the DCO approach took these metalanguages as a basis and extended on them. This property makes it more difficult to compare the two approaches. Second, the classes in TaLE used for language implementation can only be used for language parts at the level of individual production rules, whereas DCOs are particularly intended for, possibly small, groups of production rules representing a major concept in the language.

The Mjølner Orm system [Magnusson et al. 90, Magnusson 94} is an approach to object-oriented compiler development that is purely grammar-driven. Different from the traditional grammar-driven systems that generate a language compiler from the grammar, Orm uses *grammar interpretation*. The advantage of the

interpretive approach is that changes to the grammar immediately are incorporated in the language. Orm may be used to implement an existing language or for language prototyping, e.g. for application-domain specific languages. Although the researchers behind the Orm system do recognise the importance of grammar and code reuse this is deferred to future work. Extensibility and reusability are not addressed. Thus, there are several differences between the Orm approach and the DCO approach. First, Orm takes the grammar-interpretive approach, whereas DCOs extend the conventional generative approach. Second, a language implementation can be decomposed into multiple DCOs, whereas an equivalent Orm implementation would consist of a single abstract, concrete, etc. grammar, even when the size of the language implementation would justify a structural decomposition. Thirdly, the goals of the Orm system and the DCO approach are quite different. The Orm system aims at an interactive, incrementally compiling environment, whereas DCOs aim at improving the modularity and reusability of the traditional language implementation techniques.

Action semantics [Doh 94, Mosses 94] seems to do for semantic specification what the DCO approach does for language implementation, i.e. providing modularisation and extensibility. Action semantics is primarily intended for the specification of programming language semantics. It provides properties such as modularity and extensibility, so that existing specifications can be extended with additional specifications and semantic functions can be overridden by new semantic functions.

7 Conclusion

We have recognised a development away from rigid, general purpose languages towards application domain languages and extensible language models. Due to their extended expressiveness, these language models have considerable advantages over traditional programming languages. However, traditional tools and techniques for compiler construction do not provide sufficient support for the modularisation, reusability and extensibility of compiler specifications. As a solution to these problems, we have discussed the delegating compiler objects (DCO) approach and its supporting tool set, LETOS. The DCO approach provides a structural decomposition of compilers in addition to the conventional, functional decomposition. It supports modularisation and reuse of compiler specifications.

To illustrate the use of delegating compiler objects, the layered object model (**LayOM**), our experimental research language and its compiler to C++ were discussed. The resulting compiler specification was shown to be highly modular and the reuse of, for example, grammar specifications is very beneficial since it leads to small, manageable specifications. However, with respect to the problems discussed in section 2, we have experienced that some problems are not fully solved:

- **Complexity**: During the development of the **LayOM** compilers, the structural decomposition dimension provided by the DCO approach proved very useful. However, when breaking an input language into its main constructs one has to deal with the relations between these constructs. Since the language decomposition is done based on the constructs of the input language, the decomposition of the lexical and grammatical specifications does not lead to any problems. However, on the back-end side of the compiler, i.e. the code generation, the code generated for the different input language constructs may map to the same constructs in the output language. For example, in **LayOM**, some of the code generated for layers and the code generated for a **LayOM** method had to be generated as one C++ method. The synchronisation of the code generation by the various DCOs can sometimes increase complexity.
- **Maintainability**: The maintainability of the **LayOM** compiler was clearly better than in an earlier experiment with a monolithic compiler. The structural decomposition gave higher locality of reference when working on a particular language construct. However, on the back-end, maintainability is not improved for all situations. In the situation of the aforementioned overlapping code generation by several DCOs, maintainability is unchanged or, perhaps, slightly complicated.
- **Reusability and extensibility**: The properties have very much improved with the DCO approach. In our compiler development projects, reuse of existing specifications is very common. In addition, the compiler was developed in an extensible manner. First, a small base set of **LayOM** concepts was implemented

and, subsequently, the missing concepts were added as modular extensions. However, this required that the initial implementation provided for the future extension with various layer types, which formed a complicating factor.

- **Tool support**: The LETOS tool provided a considerable improvement compared to the manual approach tried by one of the students. The increased overview and support for the DCO approach proved highly beneficial. The support for code generation by the tool is limited to editing facilities for the parse graph node classes, but we are currently working on improved support.

Concluding, the lexical and grammatical support provided by the DCO approach and the supporting LETOS tool we found to be very useful. However, for semantic analysis and code generation we currently use conventional approaches that suffer from the aforementioned problems, but we intend to improve on that in the future.

We have not explicitly measured the efficiency of the resulting compilers, but we have no reason to assume that efficiency will be influenced considerably. Reused specifications are merged into the reusing specification by the tool and delegation to another DCO consists of the instantiation of a C++ object and a method call. The current **LayOM** compilers generate C++ code and in the process from a **LayOM** application to an executable program the **LayOM** compiler takes less than 5% of the time and the C++ compiler and linker the remaining 95%.

We are planning to make the Language Extensibility Tool Set, LETOS, and the **LayOM** compilers available through our ftp site. For more information, please contact the author.

Acknowledgements

The anonymous reviewers provided valuable comments that helped to improve the paper. Also many thanks to the members of the students of the Pheasant project for the construction of the Language Extensibility Tool Set (LETOS), i.e. Fredrik Hall, Jim Håkansson, Johan Ramestam, Magnus Thuresson and Piotr Gora.

Bibliography

[Bosch 95a] J. Bosch, ‘Parser Delegation - An Object-Oriented Approach to Parsing,’ in *Proceedings of TOOLS Europe ‘95*, pp. 55-68, 1995.

[Bosch 95b] J. Bosch, ‘Delegating Compiler Objects - An Object-Oriented Approach to Crafting Compilers,’ in *Proceedings Compiler Construction ‘96*, 1996.

[Bosch 95c] J. Bosch, ‘Layered Object Model - Investigating Paradigm Extensibility,’ *Ph.D. dissertation*, Department of Computer Science, Lund University, November 1995.

[Dearle 88] A. Dearle, ‘Constructing Compilers in a Persistent Environment,’ *Technical Report*, Computational Science Department, University of St. Andrews, 1988.

[Doh] Kyung-Goo Doh, ‘Action Semantics: A Tool for Developing Programming Languages,’ in Proceedings of *InfoScience ‘93*, Korea, 1993.

[Järnvall et al. 95] E. Järnvall, K. Koskimies, M. Niittymäki, ‘Object-Oriented Language Engineering with TaLE,’ to appear in *Object-Oriented Systems*, 1995.

[Magnusson et al. 90] B. Magnusson, M. Bengtsson, L.O. Dahlin, G. Fries, A, Gustavsson, G. Hedin, S. Minör, D. Oscarsson, M. Taube, ‘An Overview of the Mjølner/Orm Environment: Incremental Language and Software Development,’ *Report LU-CS-TR:90:57*, Department of Computer Science, Lund University, 1990.

[Magnusson 94] B. Magnusson, ‘The Mjølner Orm system,’ in: *Object-Oriented Environments - The Mjølner Approach*, J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, B. Magnusson (eds.), Prentice Hall, 1994.

[Mosses 94] P. Mosses, ‘A Tutorial on Action Semantics,’ Notes for *FME ‘94*, 1994.

[Pheasant 95] Pheasant student project team, ‘Pheasant project documentation,’ University of Karlskrona/ Ronneby, December 1995.

[Kuhn 62] T.S. Kuhn, *‘The Structure of Scientific Revolutions,’* The University of Chicago Press, 1962.