Class Structure Graphs for Object-Oriented Design and Implementation

Mathias Richter
Department of Computer Science
University of Zurich, Switzerland

Abstract

For software engineers, reuse in object-oriented software development requires understanding of software components written by others. At the same time it requires design of components to be reused by others. This is the reason why object-oriented software development depends on tools that support the understanding, design and reuse of software components. As a step towards such support, we introduce an approach that uses a language independent representation taking the form of graphs of object-oriented source code. Using this representation, the object-oriented design can be conducted from an abstract level to the detailed level of implementation. In turn, existing implementations can be analysed on different levels of abstraction in order to gain better understanding of a class's structure as a basis for proper reuse.
If object-oriented systems can be represented as graphs, it is possible to use graph grammars as a formal means for providing class definitions with reuse guidelines and to a certain extent reuse automation.

M. Richter
Institut für Informatik
Universität Zürich
Winterthurerstr. 190
CH-8057 Zürich
mrichter@ifi.unizh.ch
Phone: +41 1 257 4332
Fax: +41 1 363 0035

1 Introduction

The central idea of code reuse in object-oriented software development requires construction, maintenance and understanding of frameworks of classes and existing object-oriented software systems. Software engineers are constantly in a process of trying to understand the structure of software created by others, or creating more or less easily reusable software themselves. Several object-oriented design techniques have emerged [Booch94, Rumba91] to help in the design phase. Development tools [Bisch92, Gold89, Reiss90] have been created for support in the implementation phase. Design Patterns [Gamma93] provide a means for documenting the use of recurring implementation patterns in object-oriented frameworks, but lack the ability to actively guide the reuse of particular classes during design and implementation.
In this paper we introduce an approach that uses a language independent representation taking the form of graphs of object-oriented source code. We propose the use of graph productions and graph grammars as a formal means for providing class definitions with reuse guidelines and to a certain extent reuse automation. The graph representation facilitates the understanding of the structure of class definitions. In a *design process* new classes and systems of classes can be created in a step by step process, graphically editing the structure in successively more and more detail. In an *analysis process* existing classes can be viewed and examined at different levels of detail to gain understanding of a class's or a system's structure.
For the design process it must be possible to transform the graph representation of such a system into

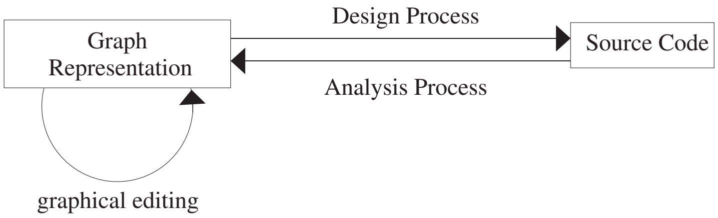


Figure 1: Designing and Analysing Classes Using Graphs.

source code of an oo-language. For the analysis process in turn it must be possible to transform source code into a graph representation (Fig. 1).
Research and practice in the application of graphs in software engineering goes back many years (e.g. SADT or SA [Ross77, DeMar78]). [Engel83, Engel89] have described a software development environment with graphs as central data structures for the programming language Modula-2, but the graphs used in their approach directly relate to the syntax graphs defining the underlying programming language. [Goedi93] uses graphs and graph grammars to describe module concepts in structured and object-oriented programming languages, focussing on the semantics of such concepts. With the advent of visual programming [Chang90], visual languages applying graph representations have been developed [Cox88, Schif94]. All these methods have in common that they are either not language independent or do not focus on the aspect of design and implementation of systems applying existing oo-programming languages like Smalltalk [Gold89], Eiffel [Meyer88] or C++ [Strou93]. In [Palsb94] class graphs are used to examine relations between classes.
We will proceed by describing constructor functions of an abstract syntax that we use to generate class structure graphs. Then we will show, how these graphs can be modified to view class structures at different levels of abstraction by expanding or collapsing certain nodes. This concludes the basic definition of class structure graphs. We then discuss the meaning of types as well as a few special configurations in our graphs. We conclude this paper by describing, how object-oriented techniques like inheritance are applied in this framework and sketching our proposal of using graph grammars to specify modifications to class structure graphs.

2 Representing Classes as Graphs

To develop a representation that is language independent, we define an abstract syntax that represents the structure of object-oriented languages like C++, Smalltalk, Eiffel, etc. We begin by explaining the construction of graphs from object-oriented source code following the analysis process of existing source code.

Programs in oo-languages are parsed according to their concrete syntax. In a second parsing process the resulting syntax tree is used to construct an abstract syntax tree according to the following abstract syntax, which we define using constructor functions. From this abstract syntax the graph representation is derived (see Fig. 2). To make this process reversible, language-dependent attributes have to be stored for each

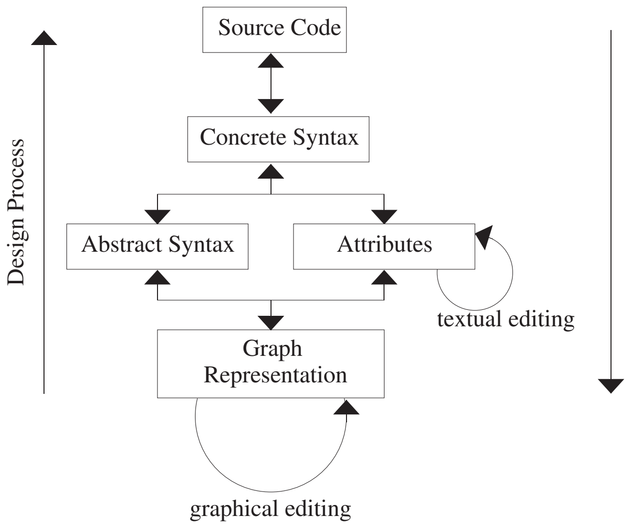


Figure 2: Obtaining Graph Representations from Abstract Syntax.

element of the abstract syntax. For now, we will not take this attribution into account. The constructor functions are defined as follows:

abs-class	:	$classname \times set(class) \times in \times out \times local \times set(method)$	$\rightarrow class$	(1)
abs-method	:	$methodname \times in \times out \times inout \times local \times list(statement)$	$\rightarrow method$	(2)
abs-call	:	$method \times in \times out \times inout$	$\rightarrow call$	(3)
abs-statement	:	$in \times out \times inout \times set(call)$	$\rightarrow statement$	(4)
abs-entity	:	$entityname \times type$	$\rightarrow entity$	(5)
abs-classtype	:	$classname$	$\rightarrow type$	(6)
abs-basictype	:	$basictypename$	$\rightarrow type$	(7)
abs-in	:	$set(entity)$	$\rightarrow in$	(8)
abs-out	:	$set(entity)$	$\rightarrow out$	(9)
abs-inout	:	$set(entity)$	$\rightarrow inout$	(10)
abs-local	:	$set(entity)$	$\rightarrow local$	(11)

Basic elements of this abstract syntax are statements, entities and types. An *entity* (5) has a name and an associated type. A *type* is either a class (6) or a basic type (7). According to their semantics (call by reference or call by value) entities are grouped into the sets *in* (8), *out* (9), *inout* (10) or *local* (11) which classify entities as input, output, input/output or local variables of a method.

A *statement* (4) has associated sets of input, output, input/output entities and a set of calls that are part of that statement. Statements form lists whose order denotes the sequential execution of the single statements of a method. In block-structured languages there can be many nesting levels of statements, which recursively form a single statement at the top-level block of a method. The statements considered here are only those in the top-level block. E.g. a case-statement in the top-level block of the source code of a method may nest further blocks within, but in our approach it forms a single statement (recursively including the nested statements).

A *call* (3) references a method to be invoked with the sets of input, output and input/output entities given to or returned from that method. A *method* (2) has a name, disjoint sets of input, output, input/output and local entities, and a list of statements. Constant number or string values in method bodies are considered as entities, too. They are entities named after the values they represent and belong to the set *local* of a method. If a method is declared as a function with a return value, an additional output entity is created that is named after the method.

Finally, a *class* (1) has a name, possibly a set of ancestors (i.e. other classes from which it inherits certain structure) and disjoint sets of input, output, and local entities (that form the *attributes* of the class) together with a set of methods. Input attributes are used to construct the internal state stored in the local attributes on creation of an object. Output attributes contain some final state when the object has accomplished its purpose. For a method or class, members of *in* cannot be modified (call by value semantics) from inside the method or class, members of *out* cannot be read from inside the method or class, members of *local* cannot be written or read from outside the method or class.[1]

The elements of the sets *class*, *method*, *statement*, *entity* and *type* produced by the described constructor functions form the nodes of class structure graphs. The visibility of certain kinds of nodes depends on the level of abstraction the graph is looked at. At the level of most detail a class structure graph consists of statement, entity and type nodes. Statement nodes are connected by *controlflow* within one list of statements. Controlflow edges to statements of another list of statements denote *method calls*. Entity nodes connect to statement nodes via *dataflow* edges. A dataflow from an entity to another entity is induced by a statement and is denoted by an edge from the "delivering" entity to the statement and an edge from the statement to the "receiving" entity.

Type nodes are connected to entity nodes by *annotation* edges to denote the flow of *type information* to an entity.

The groupings of statement, entity and type nodes that classes, methods, statement lists or sets of methods resp. entities represent, denote a change of the level of abstraction a graph is looked at. For example, a grouping of sets of entities and a set of methods can be collapsed into a node representing the according class.

The different nodes of a graph are labelled with their respective names, except for the statement nodes which are uniquely labelled "statement". The names are elements of a universe of defined names w.r.t. a language environment consisting of an object-oriented language, a class library and possibly the source code of a particular system (this complies with the closed-world assumption of [Palsb94]). The set of defined names consists of elements from the given sets *classname*, *methodname*, *entityname*, and *basictypename*.

After this informal description, the definition of class structure graphs (CSG for short) can be given:

Definition 2.1 (Class Structure Graph)

The set of nodes is $N \subseteq (\mathit{entity} \cup \mathit{statement} \cup \mathit{type})$. The set of edges is $E \subseteq (\mathit{dataflow} \cup \mathit{controlflow} \cup \mathit{annotation})$, where

$$\begin{array}{rcl} \mathit{dataflow} & \subseteq & (\mathit{entity} \times \mathit{statement}) \cup (\mathit{statement} \times \mathit{entity}) \\ \mathit{controlflow} & \subseteq & (\mathit{statement} \times \mathit{statement}) \\ \mathit{annotation} & \subseteq & (\mathit{type} \times \mathit{entity}) \end{array}$$

The set of node colors is $C_N \subseteq \mathit{entityname} \cup \mathit{basictypename} \cup \{\text{“statement”}\}$, the set of edge colors C_E is empty. ■

Let us consider a few examples. Suppose there exists a method `average` which is defined as

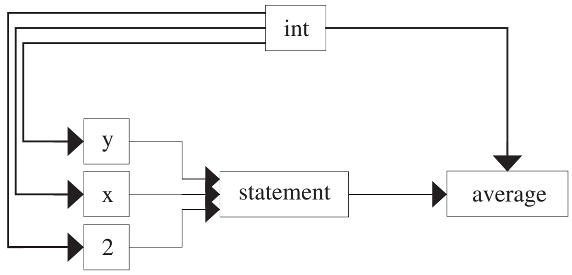
[1] We refer to Section 6.1 for a detailed discussion of encapsulation of attributes in our approach.

```
method average(x, y:INTEGER):INTEGER
begin
    average:=(x+y)/2;
end;
```

The constructor functions can be used to describe the different elements:

int-node	=	abs-basictypename("INTEGER")
x-node	=	abs-entity("x", int-node)
y-node	=	abs-entity("y", int-node)
const-node	=	abs-entity("2", int-node)
average-node	=	abs-entity("average", int-node)
stmt-node	=	abs-statement(abs-in({x-node, y-node, const-node}), abs-out({average-node}), abs-inout({}), abs-calls({}))
method-node	=	abs-method("average", abs-in({x-node, y-node}), abs-out({average-node}), abs-inout({}), abs-local({const-node}), list(stmt-node))

The corresponding graph looks like this[2]:

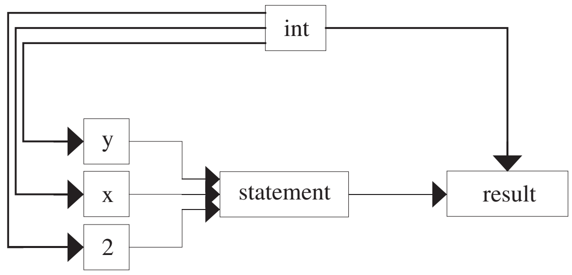


If average is not declared as a function, but instead has a parameter which on return contains the average,

```
method average(x, y:INTEGER; var result:INTEGER)
begin
    result:=(x+y)/2;
end;
```

then int-node, x-node , y-node and const-node remain the same. Further constructor functions define or redefine the elements of the resulting graph:

result-node	=	abs-entity("result", int-node)
stmt-node	=	abs-statement(abs-in({x-node, y-node, const-node}), abs-out({result-node}), abs-inout({}), abs-calls({}))
method-node	=	abs-method("average", abs-in({x-node, y-node}), abs-out({}, abs-inout({result-node}), abs-local({const-node}), list(stmt-node))



[2] The thin edges represent dataflow, the thick edges represent annotations.

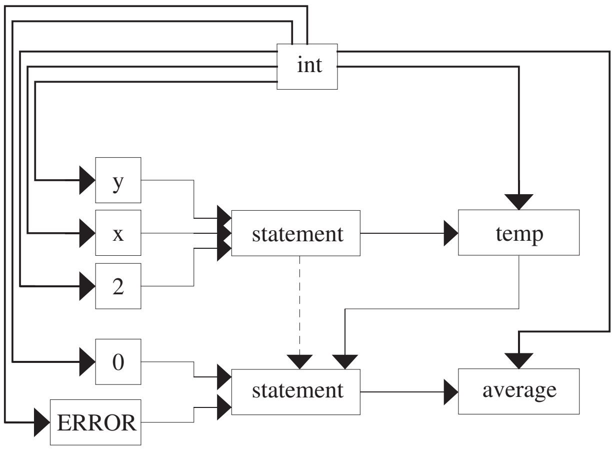
The structural identity between both graphs shows that semantically nothing has changed. Only the fashion in which the average is returned has changed.
Finally, if average is refined such that it only returns positive averages and otherwise an error constant,

```
method average(x, y:INTEGER):INTEGER
var
    temp:INTEGER;
begin
    temp:=(x+y)/2;
    if (temp ≥ 0) then
        average:=temp
    else
        average:=ERROR;
end;
```

x-node, y-node, average-node, const-node and int-node remain the same while the new elements are defined by the application of the constructor functions:

const2-node	=	abs-entity("0", int-node)
const3-node	=	abs-entity("ERROR", int-node)
temp-node	=	abs-entity("temp", int-node)
stmt1-node	=	abs-statement(abs-in({x-node, y-node, const-node}), abs-out({temp-node}), abs-inout({}), abs-calls({}))
stmt2-node	=	abs-statement(abs-in({temp-node, const2-node, const3-node}), abs-out({average-node}), abs-inout({}), abs-calls({}))
method-node	=	abs-method("average", abs-in({x-node, y-node}), abs-out({average-node}), abs-inout({}), abs-local({temp-node, const-node, const2-node, const3-node}), list(stmt1-node, stmt2-node))

The resulting graph is[3]:



3 Class Structure Graphs at Different Levels of Abstraction

Real-world systems consist of dozens or hundreds of classes, which cannot be displayed at the level of greatest detail on limited display space. It must be possible to collapse statement lists and entities grouped

[3] The controlflow edges are displayed as dotted lines.

in a method into a single node representing that method, or accordingly entities and methods into a class node. At the same time, it must be possible to expand nodes representing classes and methods into more detail as necessary.
To be able to collapse or expand class and method nodes, Definition 2.1 of Class Structure Graphs has to be changed in the following way:

Definition 3.1 (Extended Class Structure Graph)
We define a set *grouping* $\subseteq$ *class* $\cup$ *method*. The set of nodes of a CSG is extended to be $N \subseteq (entity \cup statement \cup type \cup grouping)$. The set of edges is $E \subseteq (dataflow \cup controlflow \cup annotation)$, where

$$\begin{aligned} dataflow &\subseteq (entity \times statement) \cup (statement \times entity) \cup \\ &\quad (entity \times grouping) \cup (grouping \times entity) \\ controlflow &\subseteq (statement \times statement) \cup (statement \times grouping) \cup \\ &\quad (grouping \times statement) \cup (grouping \times grouping) \\ annotation &\subseteq (type \times entity) \cup (type \times grouping) \end{aligned}$$

The set of node colors is $C_N \subseteq classname \cup methodname \cup entityname \cup basictypename \cup \{\text{“statement”}\}$, the set of edge colors C_E is empty. ■

This extended definition of CSGs introduces method and class nodes labelled with their respective names. The definition of the set of edges allows connections between the different kinds of nodes. Dataflow edges are allowed between statements, classes and methods via entity nodes. A dataflow to or from a class or method node denotes a dataflow to or from some node that was collapsed into the single method or class node.
Controlflow edges are allowed between statements, classes and methods. Controlflow from or to a class node means that a method that was collapsed with others into the single class node calls or is called from outside the class. Controlflow from or to a method means that the first statement of the statement list that was collapsed into the single method node calls or is called from outside the method.
An annotation node from a type to a class or method node means that some entities that were collapsed into the class or method node are declared with that type.
When a grouping of entities, statements and types is collapsed into a single node *all edges and nodes within* the grouping are replaced by the single node, while *all edges between nodes outside and nodes inside the grouping* are connected to the single node.
This definition of the set of edges allows only one edge of the same kind between the same two nodes. Thus, collapsing groups of nodes automatically reduces the number of edges between different collapsed groups.

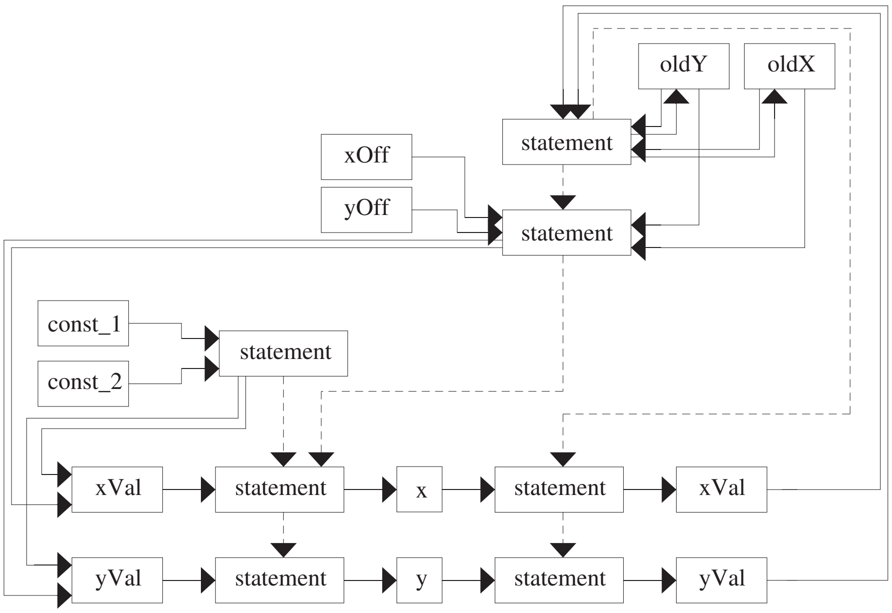
3.1 An Example CSG

As an example we will consider the following class definition of a `Point` class with attributes `x`, `y` and methods `Init`, `Set`, `Get` and `Offset`. The definition is listed below and uses a Pascal-like notation that should be self-explanatory.

```
class Point
    var x, y :  Int;
    method Set(xVal:Int; yVal:Int)
        begin
            x := xVal;
            y := yVal;
        end;
    method Get(var xVal:Int; var yVal:Int)
```

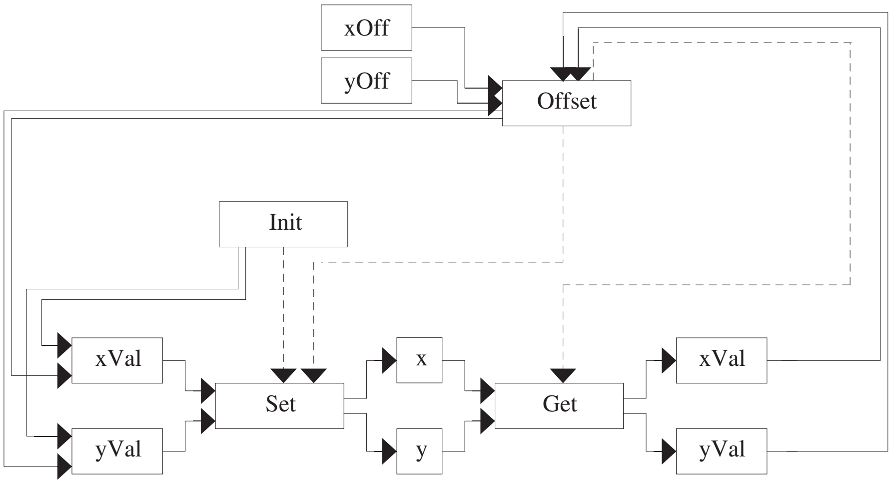
```
        begin
            xVal := x;
            yVal := y;
        end;
    method Init
        begin
            Set(0, 0);
        end;
    method Offset(xOff:Int; yOff:Int)
        var oldX, oldY:Int;
        begin
            Get(oldX, oldY);
            Set(oldX+xOff, oldY+yOff);
        end;
end class Point.
```

Again, the constructor functions of the abstract syntax are used to generate the different nodes of the CSG for the Point class (they are not shown here). The resulting graph at the most detailed level is[4]:

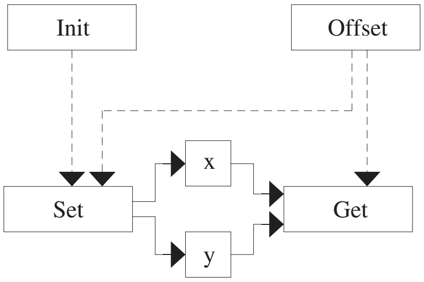


Collapsing the method bodies so that only the interfaces of the different methods with their input and output parameters are shown together with the attributes the methods read or write results in the following graph:

[4]The controlflow edges are displayed as dotted lines.



Collapsing the input and output parameters of the methods such that the graph only displays the existing methods and their connection to the attributes is shown in this graph:



Collapsing this graph into the `Point` class node is the highest abstraction. Collapsing or showing other parts of the CSG is also possible. E.g. the type nodes could be expanded out of a single class node to see which other classes or basic types are used within. The reader may have noticed that we did not include an integer type node in the CSG of the `Point` class. There is one node *int* with connections to all entities in the graph. It was left out for reasons of simplicity.
We have shown the *analysis process* in this example and have obtained a fairly complicated graph structure at the level of most detail. The question arises why to exchange an understandable textual representation by another complicated representation. The analysis process is needed in order to transform existing source code into graph representations. To understand an existing class, a programmer or designer usually approaches its structure the other way around. Starting point is the CSG at the highest level of abstraction, showing the methods and attributes with their connections. Then certain methods are expanded step by step to understand their interaction and the involved entities. The programmer will rarely look at the whole graph in full expansion. He will rather expand certain areas of the graph to study them. Being able to approach a class's structure from a high level of abstraction instead of being forced to deal with the textual representation at a steady level of detail is the advantage of CSGs.
Another advantage is the ability to conduct the design of a class on changing levels of detail. The *design process* using CSGs also starts at a level of high abstraction. The designer sketches attributes and methods with their approximate dataflow and controlflow connections. Then, step by step he refines the structures at levels of more detail and then completes statements to match the designed structure. In an imaginative development environment applying this approach, the designed structure can automatically be matched with the textual supplements (that are stored in additional attributes of the abstract syntax). The ability to decide when he wants the graph structure to be matched against the attributions needed for a transformation into actual source code, lets the designer completely integrate the phases of oo-design and -implementation.

4 The Notion of Types in Class Structure Graphs

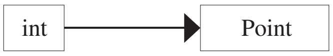
As typing is an important step towards program correctness, in our approach the usage of types is advocated. For support of untyped object-oriented languages this type information can be viewed as additional documentation.
In typed object-oriented programming languages, the annotation of an entity with a type name is purely syntactical. Any clustering of entities with a type name in a declaration does not say anything about a semantical relation between the named entities contained within. In the source code of above `Point` class example all entities have the same type, but are grouped according to syntactical conventions in PASCAL within the declarations. This is different in the CSG. If the type node *int* had been included in the graphs of the `Point` class, it would have been one node with an arc to each entity typed "Integer". There could have been one type node for each entity. But all entities of the `Point` class are dependent on a single type node, because if the representation of the coordinate system changes, e.g. to *real*, all entities must change their type accordingly (in this particular implementation). We capture this observation in the following definition:

Definition 4.1 (Type Cluster)
A *type cluster* $\mathcal{T}$ is a subgraph of a CSG $\mathcal{C} : \mathcal{T} \subset \mathcal{C}$. The set of nodes $N_{\mathcal{T}}$ consists of one type node $t \in N_{\mathcal{C}} \cap type$ and a set $ENT \subset N_{\mathcal{C}} \cap entity$, whose members e_i are in the following (symmetric, reflexive and transitive) relation:

$$e_1 \sim e_2 \Leftrightarrow (t_1, e_1) \wedge (t_2, e_2) \Rightarrow t_1 = t_2 = t$$

The sets of color alphabets of the type cluster are the respective subsets of the sets of color alphabets of the CSG. All named entities represented in the type cluster are *semantically dependent* on the type. ■

A type cluster consists of one type node and at least one node representing an entity of the CSG with edges from the type node to the entities. Type clusters are similar to Meyer's notion of *like types* [Meyer88], with the difference that in our approach *every* type is exchangeable.
The decision of how to form type clusters is a design decision made by the designer of a class. If source code is transformed into a CSG representation, the clustering of more than one named entity to a type node will have to be done by a designer knowing the semantics of the particular classes.
According to the notion of exchangeable types in class definitions in [Palsb94], the need for explicitly parameterized classes is overcome.
In the `Point` class example, there is one type cluster for the basic type *int*. At the highest level of abstraction the class is represented by the class node. Expanding the type nodes of the CSG would result in the following graph:



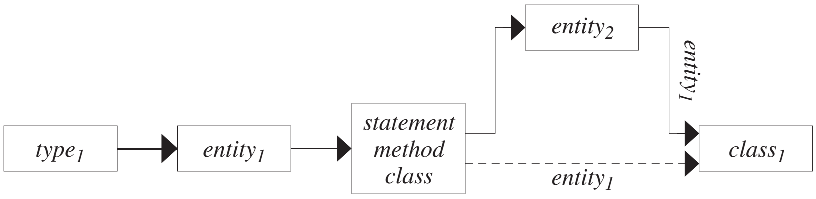
This graph shows that there is one type cluster within `Point`. It does not show which entities are involved in the type cluster unless further levels of detail are revealed. Using the information displayed in the above graph, a programmer is able to convert the integer implementation of the `Point` class into a real implementation by exchanging the type node *int* by a type node *real*. Checking whether this exchange matches with the operations performed on the entities of the type cluster within the statements of the class's methods can be automated in a software development environment, because all types have to be part of the known language-environment. If inconsistencies are detected, they can be pointed out to the programmer for correction. In the case of the `Point` class, the *int* to *real* conversion can be done without any problems, because the only operation performed on the entities is addition, which is valid for both types.

4.1 Class Types

By allowing the use of classes as types[5] object-oriented languages support the construction of complex classes, i.e. classes as parts of other classes.
Consider the following class `foo`, which has an attribute `entity1` of type `class1` and a method `bar` with a local entity `entity2` of some type `foobar`. The method `bar` calls some method of `class1` using the reference `entity1`:

```
class foo
    var entity1 :  class1;
    method bar
        var entity2 :  foobar;
        begin
            entity1.method_of_class1(entity2);
        end;
end class foo.
```

The graph for such a structure could have the following general appearance:



The type node $type_1$ represents the class type constructed from $class_1$. The controlflow or dataflow edges to the node $class_1$ are labelled $entity_1$ to show that `entity1` serves as a reference to `class1`. We have to extend Definition 3.1 to include the color alphabet for controlflow edges:

Definition 4.2 (Complete Class Structure Graph)
The sets of nodes, edges and node colors follow Definition 3.1. The set of edge colors is $C_E \subseteq entityname$. C_E is mapped by a partial function to the set of edges E. ■

Above graph represents the *forwarding* technique which is widely used to implement code reuse without inheritance in languages where inheritance is subtyping.

5 Structural Observations in Class Structure Graphs

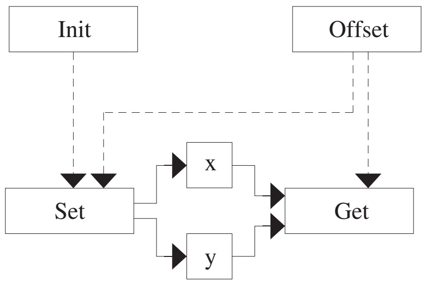
The structural complexity of a CSG characterizes a class in another way than by means of number of methods and attributes or lines of code in methods.
Nodes of a CSG have a *fan-in* and a *fan-out* just like elements of electronic circuits. The *data fan-in/out* is the number of dataflow edges to resp. from a node. The *control fan-in/out* is the according number of controlflow edges to resp. from a node.
We characterize nodes by their data and control fan-in and fan-out. A node with a higher data fan-in than fan-out is called *concentrator*. A node with a higher data fan-out than fan-in is called *distributor*. Nodes with equal data fan-in and fan-out are called *connector*. Accordingly, we call nodes with a higher control fan-in than fan-out *passive* resp. *active* in the reverse case.
Attributes store the state of an object and methods are used to alter this state. From a puristic point of view, methods that do not read or write any attributes of their class produce side effects. We call such

[5] Independently of whether the type hierarchy is or is not identical with the inheritance hierarchy.

	entity	*statement*	*method*	*class*	*type*
control fan-in/out		•	•	•	
data fan-in/out	•	•	•	•	
concentrator	•	•	•	•	
distributor	•	•	•	•	•
connector	•	•	•	•	
active		•	•	•	
passive		•	•	•	
detached			•		
clustering ratio	•				•

Table 1: Overview of Terms and Measures in Class Structure Graphs.

methods *detached methods*.The number of detached methods in a class reveals the importance of an object's state for its purpose. A class that consists of detached methods only could, for example, be one that just groups a number of related auxiliary operations instead of pursuing a particular purpose during its lifetime. Another way to look at detached methods is that they represent an interface of a class, shielding methods that directly alter the object state from outside access.
In this context the number of active or passive methods allows a programmer to draw conclusions about a class's central methods. Methods that are not detached tend to be passive, because they are called more often simply to change an attribute's value by detached methods.
The number of type clusters is also an interesting measure for the complexity of a class's structure. The *clustering ratio* is computed as the ratio of the number of entity nodes versus the number of type nodes. The smaller this ratio is, the clearer is the class's design w.r.t. types. The reciprocal clustering ratio is the average number of entities included in the type clusters of a CSG. In order to be useful measures, both values have to be viewed in relation to the total number of entities in a CSG.Table 1 gives an overview of the discussed terms and the involved nodes. As an example, we consider the collapsed CSG of the `Point` class again:



The methods `Init` and `Offset` are detached active methods in this class (although their actions indirectly result in state changes), while `Set` and `Get` are passive methods. Clustering ratio and data fan-in/fan-out are not visible at this level of abstraction. As mentioned in Section 4, the CSG of the `Point` class includes only one type cluster which consists of the type node *int* connected to every entity, which makes the clustering ratio very small and shows that this implementation only includes the most necessary entities.

6 Object-Oriented Structuring Techniques with CSGs

Encapsulation, abstraction and inheritance are the important concepts in the object-oriented paradigm [Booch94]. In this section we sketch how they are supported by CSGs in our approach.

	Level 1	Level 2	Level 3
Attributes	all visible	all invisible, access through methods	visible for object only, for all objects of one class or completely visible
Methods	all visible	all visible	visible for object only, for all objects of one class or completely visible

Table 2: Visibility of Attributes and Methods in OO-Languages.

6.1 Encapsulation

Encapsulation in general hides an object's state from access by other objects. Basically, there are three levels of visibility supported by different oo-languages (Table 2). Although it is very useful to have several levels of visibility, it is often impossible to completely foresee the use of a class and decide on which attributes and methods are visible and which are not — especially when building a class at a very high level of abstraction. Usage of constructs like the "friend" construct in C++ [Strou93] to circumvent access restrictions requires modification of other classes where the visibility decision was made and results in unintentional relations between classes. Furthermore, most visibility support does not take the lifecycle of an object into account. On creation, an object gets certain initial data which is used to construct an internal state maintained during the lifecycle. On deletion, the object has a certain final state which might be of interest for other objects.
In a CSG, a class's attributes are grouped in three disjoint sets, which can be seen from the constructor function of a class in our abstract syntax:

$$\text{abs-class} : \textit{classname} \times \textit{set}(\textit{class}) \times \textit{in} \times \textit{out} \times \textit{local} \times \textit{set}(\textit{method}) \rightarrow \textit{class}$$

The three sets *in*, *out* and *local* make up the attributes of a class. The entities of *in* can be written from outside the class, but can only be read by the class itself. They are "channels" through which values can be delivered to the class. The entities of *out* can only be read from outside the class and are written from inside the class. The entities in *local* are not accessible from outside the class itself, because they store the actual state of the class.
According to the constructor function for a class there is only one set of methods which we define to be accessible from in- and outside the class. However, further development of this approach might necessitate the introduction of a set of *local methods* which can only be accessed from inside a class. This can easily be done by introducing another constructor function in the abstract syntax that groups disjoint sets of methods into local or "regular" methods.

6.2 Abstraction

In the object-oriented paradigm abstraction is supported by two different mechanisms. *Abstract classes* are classes that do not implement certain methods. Such classes have to be subclassed where the unimplemented methods have to be implemented.
In a CSG, abstract classes can be constructed with methods specified at different levels of detail. In all cases the statement list of an abstract method is empty. It is possible to specify a method only by its name without requiring a certain signature at all, or the full signature can be specified with an abstract method. The abstract method's node cannot be expanded into statements, because the statement list is empty.
The *genericity* of classes increases their reusability in different contexts. Container classes that implement lists, stacks, sets, etc. provide the classic area of application for generic classes. A generic class implements

a certain functionality that applies to different types or classes. Lists, for example, can store anything from integer values to complex objects. So it must be possible to implement the functionality of a list once and specify the type of elements stored within from case to case. Section 4 explained that basically every type node can be exchanged so that the need for an explicit mechanism is overcome.

6.3 Inheritance

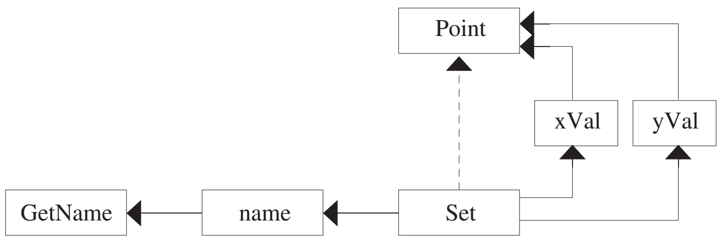
Inheritance as one of the main concepts of the object-oriented paradigm allows successors of a class to inherit attributes and code from their ancestors. Inheritance results in the following possible modifications to a class:

- addition of attributes
- overriding of methods
- addition of methods
- implementation of abstract methods

In our approach, each class can have a set of ancestors which is represented in the graph by collapsed class nodes. Consider a subclass of `Point` which additionally stores a name for each point, overrides the method `Set` to additionally set the point's name and adds a method `GetName` that returns the point's name:

```
class NamedPoint
    inherits Point;
    var name :  String;
    method Set(xVal:Int; yVal:Int; theName:String)
        begin
            inherited Set(xVal, yVal);
            name := theName;
        end;
    method GetName:String
        begin
            GetName := name;
        end;
end class NamedPoint.
```

The following CSG represents the new situation. The node "Point" is the class node of `NamedPoint`'s ancestor and receives dataflow from the overridden method `Set` which calls inherited `Set`.



Expanding the class node of the superclass `Point` would reveal the call to the inherited method `Set`. Note that the two connectors `xVal` and `yVal` are the input entities of the inherited method `Set` in the `Point` class.
A successor of a class can either be a subclass or a subtype. A subclass merely reuses structure of its ancestor without being a subtype, i.e. it cannot be used in every context a subtype could be used. When creating a class by inheritance, the designer must specify whether he wants that class to be a

subclass or a subtype. When the CSG representation is transformed back into source code of an actual language, the subtyping or subclassing has to be implemented according to that language's support for subtyping/subclassing. In languages where inheritance is subtyping, subclasses that are not subtypes have to be implemented using the aforementioned technique of *forwarding*. Usually, modifications are induced by *monotonic* inheritance [Wegne88]. Using a CSG representation, consequences of nonmonotonic inheritance (i.e. deletion of attributes and methods of a superclass in the subclass) can be studied further.
Note that CSGs are not intended to replace inheritance trees. On the contrary, the nodes of an inheritance tree are collapsed class nodes representing CSGs at the highest level of abstraction. Thus, expanding single nodes of the inheritance tree into detailed CSGs reveals another dimension of the tree. A software development environment could provide for a three-dimensional visualization of this context.

7 Conclusion and Future Work

In the previous sections we have explained the representation of object-oriented systems as graphs with the consequences concerning the notions of type, inheritance and encapsulation. The motivation was the possibility of analysing existing classes and designing new classes on different levels of abstraction *and* the visualization means that graphs provide. In combination with at the same time powerful and flexible visualization, the usage of graphs has another advantage: The possibility to formalize modifications of graphs with graph grammars, in analogy to the modification of strings by context-free grammars. In this concluding section we sketch a few starting points for the usage of graph grammars with CSGs for oo-design and -implementation, which is main part of our current research.
In [Wegne88] Wegner describes the mechanism of inheritance as

> *"[...] a particular kind of incremental modification that transforms a parent entity P with a modifier M into a result entity R = P + M."*

The modification of a CSG by inheritance as described in the previous section results in changes in the graph's structure. These changes can be described by graph productions of a graph grammar.
A production of a graph grammar describes the replacement of one subgraph of a graph by another subgraph. A graph grammar takes a start symbol and applies all productions to a graph until no further replacements can be made. A complete survey of graph grammars and the implications is given in [Ehrig79, Nagl79].
If the parent entity P is a CSG, then the modifier M achieving incremental modification into another CSG R = P + M is a graph production or a whole graph grammar. The addition operator can then be interpreted as "deriving from P using M", or $P \Longrightarrow_M R$.
Class hierarchies as they are implemented and documented today, demand a considerable learning effort for a designer/programmer to apply them in the development of object-oriented systems. Often enough insufficient understanding of a class hierarchy results in classes being modified improperly, or classes being used in wrong contexts.
The CSG representation of a class's structure enables the designer of a class to provide modifiers in the form of graph productions. Other programmers using a class in their systems can apply these modifiers and complete the resulting structure with the functionality needed in the particular context.
This means providing classes to a certain extent with *reuse intelligence*. It enables a software development environment to *actively guide* code reuse. In such an environment the designer of a class could even describe the viewing operations on a CSG (i.e. collapsing and expanding nodes) by graph productions and thus specify what others may or may not see when browsing the class.
Furthermore, the representation of CSGs on computer displays is not limited to the graphs used in this work. One could imagine three-dimensional or other visualizations [Sarka93, Koike92], even animation of class modifications.

8 Acknowledgements

The author is grateful to Peter Baumann, Clemens Cap and Helmut Schauer for comments, discussions and useful suggestions on this work.

References

[Bergs89] J. A. Bergstra, J. Heering, P. Klint (Eds.): *Algebraic Specification*, ACM Press, 1989.

[Bisch92] W. R. Bischofberger: *Sniff — A Pragmatic Approach to a C++ Programming Environment*, in Usenix C++ Proceedings, Usenix Association, 1992.

[Booch94] G. Booch: *Object-Oriented Analysis and Design with Applications*, 2nd Edition, Benjamin Cummings Publishing, 1994.

[Chang90] S.-K. Chang: *Principles of Visual Languages*, in Principles of Visual Programming Systems, Prentice Hall, 1990.

[Cox88] P. T. Cox, T. Pietrzykowski: *Prograph: A Visual Dataflow, Object-Oriented Software Engineering Environment*, Technical Report, Technical University of Nova Scotia, 1988.

[DeMar78] T. DeMarco: *Structured Analysis and System Specification*, Yourdon Press, 1978.

[Duby92] C. K. Duby, S. Meyers, S. P. Reiss: *CCEL: A Metalanguage for C++*, Technical Report CS-92-51, Brown University, 1992.

[Ehrig79] H. Ehrig: *Introduction to the Algebraic Theory of Graph Grammars*, in Graph Grammars and Their Application to Computer Science and Biology, LNCS 73, Springer, 1979.

[Engel83] G. Engels, R. Gall, M. Nagl, W. Schäfer: *Software Specification Using Graph Grammars*, in Osnabrücker Schriften zur Mathematik, Reihe Informatik, FB Mathematik der Universität Osnabrück, No. 9, 1983.

[Engel89] G. Engels, W. Schäfer: *Programmentwicklungsumgebungen – Konzepte und Realisierung*, Teubner, 1989.

[Gamma93] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, in ECOOP '93 Proceedings, O.-M. Nierstrasz (Ed.), Springer, 1993.

[Goedi93] M. Goedicke: *On the Structure of Software Description Languages: A Component Oriented View*, Technical Report No. 473, Universität Dortmund, 1993.

[Gold89] A. Goldberg, D. Robson: *Smalltalk-80: The Language*, Addison-Wesley, 1989.

[Koike92] H. Koike: *An Application of Three-Dimensional Visualization to Object-Oriented Programming*, in Proceedings of the International Workshop on Advanced Visual Interfaces '92, T. Catarci, M. F. Costabile, S. Levialdi (Eds.), World Scientific, 1992.

[Meyer88] B. Meyer: *Object-Oriented Software Construction*, Prentice Hall, 1988.

[Nagl79] M. Nagl: *Graph-Grammatiken*, Vieweg, 1979.

[Palsb94] J. Palsberg, M. I. Schwartzbach: *Object-Oriented Type Systems*, J. Wiley & Sons, 1994.

[Reiss90] S. P. Reiss, S. Meyers: *FIELD Support for C++*, in Usenix C++ Proceedings, Usenix Association, 1990.

[Ross77] D. T. Ross, K. E. Schoman jr.: *Structured Analysis for Requirements Definition*, IEEE Transactions on Software Engineering, Vol. 3, No. 1, January 1977.

[Rumba91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[Sarka93] M. Sarkar, M. H. Brown: *Graphical Fisheye Views*. Technical Report CS-93-40, Brown University, 1993.

[Schif94] S. Schiffer, J. H. Fröhlich: *Concepts and Architecture of VISTA - a Multiparadigm Programming Environment*, in Proceedings of IEEE Symposium on Visual Languages, 1994.

[Strou93] B. Stroustrup: *The C++ Programming Language*, 2nd Edition, Addison Wesley, 1993.

[Wegne88] P. Wegner, S. B. Zdonik: *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like*, in ECOOP/OOPSLA '88 Proceedings, S. Gjessing, K. Nygaard (Eds.). Springer, 1988.