Subtyping and Application of Context-Free Classes

Welf Löwe, Jürgen Freudig, Rainer Neumann, and Martin Trapp
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
76128 Karlsruhe, Germany.
E-mail:{loewe|freudig|rneumann|trapp}ipd.info.uni-karlsruhe.de

December 2, 1997

Abstract

Object-oriented systems are usually not designed from scratch but constructed using frameworks or class libraries. This construction should lead to correct systems provided the reused classes are locally correct. Therefore knowledge about the features that a certain class provides is often not enough. It is additionally necessary to know the correct semantics of classes, i.e. information on how to use these features. Especially, we have to mind the sequences of method calls that are acceptable by an object. Using regular languages for the description of these sequences works fine for some classes but is not adequate for others, e.g. for *stacks*, *buffers*, *queues* or *lists* etc.

In this paper we present a more general approach for the specification of the dynamic behavior of objects using context-free grammars. We investigate questions of correctness in subtyping and application of such classes. We define sufficient conditions for class systems such that local correctness in subtyping and application implies global correctness of the system.

1 Introduction

The cost effective construction of high quality software is the main issue in software engineering. It requires the correct composition of components. The object-oriented paradigms promised to solve this problem, e.g. [2, 33]:

- Classes define software components with clearly defined interfaces providing reusable services.
- Abstract classes allows implementation of frameworks that contain general solutions on an abstract level. These frameworks can be instantiated to application domain specific system. Polymorphism provides the base for

this construction. Objects that provide a certain interface can be plugged into abstract solutions in order to build concrete systems.

- Inheritance allows to reuse, mix or adapt the behavior (or only the implementation) of other classes.

In practice, there are many problems with using these techniques in system design: locally correct classes may lead to globally incorrect systems – systems are not safe by construction.

GAMMA's *Design Patterns* [14] represent expert knowledge on the design of reusable components (class structures). His work contains agreed expert solutions to several design problems. The informal description may lead to errors in construction. Especially, the missing specification of the dynamic behavior is missing or poor. Most of the discussions in there are about flexibility rather than correctness. However, the goal should be the design of correct systems preserving flexibility.

MEYER's *Design by Contract* [25] is one step towards this goal. The *software contracts* describe the interactions between objects by pre- and postconditions of methods and class invariants. But still the use of inheritance, polymorphism or genericity may lead to incorrect systems – the global system correctness may be destroyed by locally correct classes.

WEGNER's and ZDONIK's *Principle of Substitutability* [40] requires that instances of subtypes can always be used in any context in which an instance of a supertype was expected. This guarantees the safe reuse of class structures even if components, i.e. classes, are substituted by other classes.

AMERICA showed that *contra-covariance* of pre- and postconditions of methods implies substitutability of these classes [1]. It is required that (i) for each method m_{super} in the superclass there exists a method m_{sub} in the subclass with the same name, (ii) the pre-conditions of methods m_{super} imply the pre-conditions of the methods m_{sub}, and (iii) the postconditions of m_{super} may be followed from the post-conditions of the methods m_{sub}.

LISKOV and WING relaxed this definition while preserving the substitutability [22]. They prove that (iii) must only satisfied for those post-conditions of a subclass that may become true in the context of the superclass. ZAREMSKI and WING use a proof theoretic approach to check the conformance and the use of objects for correctness in Larch/ML [41] programs [42].

In object-oriented languages like Simula [11], C++ [38], Modula-3 [4, 5], Sather(-K) [37, 15], and Java [16], contra-covariance (or a stronger condition) is checked only for those predicates that are computable at compile time, namely for the types. This approach was formalized by CARDELLI [3] or FRICK et al. [12]. Assume, a class A' is declared a subclass (-type) of A (denoted by $A' \leq A$). A type systems only check if (i) holds. Additionally, for all corresponding methods m_{super} and m_{sub} and for parameter types P^i_{super} and P^i_{sub} it must be $P^i_{super} \leq P^i_{sub}$ (the index i denotes the parameter identification, e.g. name or/and position). This is obviously weaker than (ii). Finally, for the result types R_{super} and R_{sub} (if any) it must be $R_{sub} \leq R_{super}$ which is weaker than

(iii). To show that the dynamic behavior of an objects of a class conforms to an object of another class, proof obligations for the software engineer remain. However, there is no general approach to check these obligations by a compiler. This is not surprising because, in general, it is an undecidable problem.

Our question was the following: Is it possible to specify the dynamic behavior of objects (beyond types) such that we can statically check whether or not:

(1) a class conforms to another class, and

(2) a class is applied correctly in a certain context.

CHAMBERS defines *Predicate Classes* [6] for the language CECIL [7]. Objects of a given class may accept some of its methods only if some properties are true. Such a class is described by several property classes. These property classes correspond to states of a finite automaton and accept only a subset of the class' methods. Statically it is not decidable which transition is executed at run time. Therefore, neither correctness of application nor conformance in the sense we defined it may be tested statically.

For the programming language Eiffel [26], MITCHELL ET AL. describe how to enhance the checks of software contracts, see [27]. The evaluation is done at run time.

The work most closely related to ours is that of NIERSTRASZ [30]. He describes the dynamic behavior of objects in terms of *regular types*. The sequences of method calls that class may accept are defined by a regular language. Conformance (request substitutability) and correct application of classes (request satisfiability) are defined in terms of inclusion of the corresponding regular languages. He adopts an algorithm known from the theory of formal languages to check whether or not two classes are request substitutability. The same algorithm could be applied for the request satisfiability problem. However, therefore single objects must be considered instead of classes. Additionally, the call sequences to the single objects have to be determined. Both questions are not addressed in [30]. Additionally, if the regular sets are defined by nondeterministic automata, the test may become exponential in time.

Not all call sequences can be defined by regular sets. Therefore, our approach uses context-free grammars to narrow the dynamic behavior of objects. We apply results from formal language theory to derive a sufficient criteria for conform subtyping. This criteria is closely related to the structural containment of grammars . Structural containment of the grammars is decidable for context-free languages. We show how a slightly modified condition can be computed efficiently. Additionally, we know proper subsets of context-free languages for which the problem becomes computable in polynomial time. Furthermore, we show how a dynamic check for correctness of an application of such objects can be generated. We apply techniques that are used to generate parsers from context-free languages. We introduce an analysis that, if successful, allows to omit the runtime checks. This analysis traces the control flow of a program, generates a context-free grammar of all method calls and slices this grammar w.r.t. single instances of classes.

This paper is organized as follows: First, we give some basic definitions. Second, we define a sufficient criteria to guarantee conformance at compile time. Third, we discuss the dynamic checking of correctness in applications and define an analysis allowing to drop these runtime tests. Finally, we conclude the results and sketch our further work.

2 Basic Definitions and Observations

An object oriented class *system* σ is a collection of classes that together define an executable program. We define local correctness and conform subtyping of classes. Furthermore, we define the correct application of objects of such classes. These definitions partially base on those introduced in [1, 22].

Definition 2.1 (Local Correctness) *Let A be a class and M_A be the set of methods of A. A is locally correct iff:*

$$\begin{aligned} & inv_A \textit{ is satisfiable} \quad \wedge \\ & \exists m \in M_A : pre_{m,A} \wedge inv_A \textit{ is satisfiable} \quad \wedge \\ & \forall m \in M_A : \{pre_{m,A} \wedge inv_A\} b_m \{post_{m,A} \wedge inv_A\}, \end{aligned}$$

where $pre_{m,A}, inv_A, b_m$, and $post_{m,A}$ is the precondition of m, the invariant of A the method body of m, and the post-condition of m, respectively. $\{P\}b_m\{Q\}$ asserts that Q holds after the execution of b_m provided P holds before its execution.

Remark: Condition (1) guarantees that there exists at least one instance of A. Condition (2) guarantees that there is at least one method applicable to an instance of A. Condition (3) guarantees that the method's execution does not contradict the invariant of A. $\diamond$

Attributes are modeled by a pair of access methods. Note, that the preconditions are evaluated in terms of the state before the body of m is executed while the postconditions are evaluated in terms of the state after that execution.

Local correctness is assumed in the following. We define the notion of conformance of two classes A and A' recursively:

Definition 2.2 (Conformance) *Conformance $\sqsubseteq_c$ is reflexive and transitive. $A' \sqsubseteq_c A$ iff:*

$$\begin{aligned} & inv_{A'} \Rightarrow inv_A \quad \wedge \\ & \forall m \in M_A \; \exists m' \in M_{A'} : m' \sqsubseteq_c m. \end{aligned}$$

$m' \sqsubseteq_c m$ iff

$$\begin{aligned} pre_m \wedge inv_A \quad &\Rightarrow \quad pre_{m'} \wedge inv_{A'} \quad \wedge \\ post_{m'} \wedge pre_m \wedge inv_{A'} \quad &\Rightarrow \quad post_m \wedge inv_A. \end{aligned}$$

Conformance of methods cannot be checked statically since the simulation of Turing machines can be reduced to this problem. Therefore a weaker criteria based on the static types is used This criteria can be checked statically. Definitions based on the parameters and result types, e.g. in [3, 12], abstract from the following properties of class systems[1]:

(1) Certain methods in the system must be called in the body of m' in order to conform to m. Consider e.g. calls from m back to the system.

(2) A history of method calls may be required before m' can be executed, including (a) calls to the object containing m and (b) calls to parameters. An example is that all objects must be initialized be explicitly calling a certain method before any other method can be executed.

(3) The result is used in the class system, i.e. methods are sent to this result. Again, these methods may require other methods to be called in m.

The abstraction from (1) is admissible if there is no call from m back to objects that are known by other instances of classes in the system or if these calls have no side effects. (2) can be ignored if m can be invoked in an instance of the considered class independently of its current state. The same must be true for any method in the parameter classes called inside m. (3) does not need to be considered if, e.g., any method in the result class can be called in an arbitrary order.

This work extends the state of the art by considering (2) for the class containing m. We assume therefore that abstractions from (1), (2b), and (3) are admissible in the class system.

Example 1 *We consider an abstract data type* `stack` *which is able to capture elements of type* T*:*

```
class stack(T) is
   push(e:T);
   pop:T;
   empty:BOOL;
end class;
```

The methods are not yet implemented, so abstraction (1) is no problem. No method is called on the parameter. Hence abstraction (2b) and (3) are admissible. However, `pop` *must not be executed on an empty stack.*

We define the notion of conform subtyping as a weaker condition than conformance:

Definition 2.3 (Conform Subtyping) *Conform Subtyping* $\sqsubseteq$ *is reflexive and transitive.*

[1] Properties could be specified using pre- and postconditions.

Let A be a class and M_A be the set of methods of A. We denote M_A the alphabet *of A. Let A' be a class with alphabet $M_{A'}$ The alphabet $M_{A'} \sqsubseteq M_A$ iff*

$$\forall m \in M_A \; \exists m' \in M_{A'} : m' \sqsubseteq m.$$

$m' \sqsubseteq m$ iff

$$\begin{array}{rcll} P^i_m & \sqsubseteq & P^i_{m'} & \wedge \\ R_{m'} & \sqsubseteq & P_m. & \end{array}$$

Let h be the function that matches m to m', iff $m' \sqsubseteq m$. A class A restricts the valid sequence of accepted methods by a language $L_A(M_A)$ over the alphabet M_A. $A' \sqsubseteq A$ iff

$$\begin{array}{rcll} M_{A'} & \sqsubseteq & M_A & \wedge \\ H(L_A(M_A)) & \subseteq & L_{A'}(M_{A'}), & \end{array}$$

where $\subseteq$ is the usual inclusion of languages and $H(L)$ is the complete application of h on all methods in the sequences in L.

The type systems of programming languages like C++, Java, and Sather assume the languages over the alphabets M_A always to be the regular languages M_A^* where * is the Klenee [19] operator. Then the abstractions from (1), (2), and (3) are admissible. In this case the relations $\subseteq$ (defined above) and $\leq$ (defined in the introduction) are equal. Therefore these languages only check for the conformance of alphabets. Nierstrasz [30] describes the history of method calls by regular languages. Implicitly, he makes the same assumptions for the system as we do, i.e. he abstracts from (1) (2b), and (3). Chambers [6] also motivates other languages but cannot statically check for conformance. In contrast to these approaches, we discuss context-free languages.

In the following we use $L(A)$ and $L_A(M_A)$ as synonyms.

Example 2 *For the class in example 1, the language L is defined by the context-free grammar $G = (S, M, N, P)$ with the alphabet $M =\{$ `pop`, `push`, `empty` $\}$ the non terminal symbols $N = \{S, T\}$ and the productions P:*

$$\begin{array}{rcl} S & \rightarrow & S\ T \mid \\ & & T \\ T & \rightarrow & \texttt{push}\ S\ \texttt{pop} \mid \\ & & \texttt{push pop} \mid \\ & & \texttt{empty}. \end{array}$$

An extended stack may capture elements of type `T` *and* `T'`*:*

```
class extended_stack(T,T') is
   push(e:T);
   push'(e:T');
```

```
    pop:T;
    pop':T';
    empty:BOOL;
end class;
```

For this class, the language is defined by the context-free grammar $G' = (S', M', N', P')$ *with the alphabet* $M' =$ { `pop`, `push`, `empty`, `pop'`, `push'` } *the non terminal symbols* $N' = \{S', T'\}$ *and the productions* P'*:*

$$\begin{array}{lcl} S' & \rightarrow & S'\ T' \mid \\ & & T' \\ T' & \rightarrow & \texttt{push}\ S'\ \texttt{pop} \mid \\ & & \texttt{push pop} \mid \\ & & \texttt{push'}\ S'\ \texttt{pop'} \mid \\ & & \texttt{push' pop'} \mid \\ & & \texttt{empty}. \end{array}$$

It guarantees that `pop` *is allowed iff the top of the stack is of type* `T` *and* `pop'` *is allowed iff the top of the stack is of type* `T'`*.*

Obviously the extended stack conforms to the simple stack from example 1.

In the remainder of this section we discuss the correctness of the applications of objects and classes. Therefore we define:

Definition 2.4 (Application of Objects and Classes) *An application of an object* a *of a class* A *is the call* $a.m$ *of a method* m *in* a*. An application of a class is a declaration* $v : A$ *which defines a variable for instances of* A*.*

An application of an object is correct iff the caller guarantees the preconditions of m and accepts the postconditions of m.

Definition 2.5 *Let* $v : A$ *be an application of a class* A*. Let* $v.m$ *be an application of the object* a *contained in* v *at a certain point in the execution. Let* V *be the set of all objects possibly contained in* v*. Let* $\{P\}v.m\{Q\}$ *be the requirement for the correctness of* $v.m$ $v : A$ *is correct iff*

$$\begin{array}{lll} \forall v.m \in \sigma : \forall a \in V : & P \Rightarrow pre_{m,A} \wedge inv_A & \wedge \\ & post_{m,A} \wedge inv_A \Rightarrow Q. & \end{array}$$

Again the correctness of an application of a class cannot be tested statically in the general case. However, in [1, 22] it is shown that

$$(A' \sqsubseteq_c A) \Rightarrow (v : A \text{ is correct} \Rightarrow v : A' \text{ is correct}). \tag{1}$$

Similar results are obtained by replacing $\sqsubseteq$ by $\leq$ and assuming an application to be correct (-ly typed) iff is not possible to have a dynamic type error while executing a program, cf. [3, 12]:

$$(A' \leq A) \Rightarrow (v : A \text{ is correctly typed} \Rightarrow v : A' \text{ is correctly typed}). \tag{2}$$

Since we restricted the legal calls to an object a of class A by $L_A(M_A)$, certain calls $a.m$ may be rejected by a although the program is correctly typed. Now we are ready to define the notion of correct application of a class that may be followed from $\sqsubseteq$, analogous to the implications (1) and (2):

Definition 2.6 (Correct Application of Objects and Classes) *Let a be an object of class A. Let $L(a)$ be a language over M_A. $L(a)$ contains all sequences of calls possibly occurring in an execution of a system σ. a is applied correctly in σ iff*

$$L(a) \quad \subseteq \quad L_A(M_A),$$

Let $v : A$ be an application of a class A. Let V be the set of all objects possibly contained in V. For each $a \in V$, let $L(a)$ be the language of calls to a for executions of σ. A is applied correctly iff

$$\forall v \in \sigma : \forall a \in V : \textit{applied correctly.}$$

This definition is stronger than that used in equation (2) since it rejects also correctly typed applications because of other dynamic errors that might occur. It is weaker than the notion of correctness from equation (1).

Theorem 2.7 *Let $v : A$ be an application of a class A in σ and $A' \sqsubseteq A$. If $v : A$ is correct then $v : A'$ is also correct.*

Proof: The first part of the proof should show that for each individual call $a.m$ the call $a'.m'$ does not cause a type error. This is obviously true since $M(A') \sqsubseteq M(A)$. For details we refer to [3, 12].

Now we prove that no call $v.m'$ is rejected in a sequence $v.m'_1, \ldots, v.m'_n$, of calls if the sequence $v.m_1, \ldots, v.m_n, h(m_i) = m'_i$, was accepted:

Assume the opposite was true, i.e. it exists a sequence of calls $m_1, \ldots, m_n$, such that a call of m is accepted next by a and after the call sequence $m'_1, \ldots, m'_n$, a call of m' is rejected by a'. The sets of all legal call sequences of the class A and A', respectively, is defined by the sets of all prefixes of legal sentences $l \in L(A)$ and $l' \in L(A')$, respectively. The class A' conforms to A, i.e. the set of legal sentences $l \in L(A)$ is a subset of legal sentences $l' \in L(A')$. This implies that the same is true for the prefixes of both languages which is in contradiction to the assumption above. $\diamond$

3 Conformance

In general it is not decidable for two context-free languages L and L' whether or not $L \subseteq L'$. However, we are able to derive sufficient criteria to guarantee $L \subseteq L'$. One sufficient criterion uses *parenthesis grammars* introduced by McNaughton in [24].

A context-free grammar $(G) = (S, M \cup \{(,)\}, N, P)$ is a parenthesis grammar iff the right-hand side of any production is of the form $X \to (w), w \in (M \cup N)^*$. (G) is called the parenthesis version of the grammar $G = (S, M, N, P)$ if for each production $X \to w, w \in (M \cup N)^*$, (G) contains a production $X' \to (w')$, with $X = X'$ and $w = w'$. For parenthesis languages the equivalence problem is decidable [24]. If the parenthesis versions of two grammars are equivalent, the grammars generate the same words with the same form of derivation tree [34]. Obviously it holds

Lemma 3.1 *If the parenthesis versions (G_1) and (G_2) of two grammars G_1 and G_2 are equivalent then for the induced languages it holds that $L(G_1) = L(G_2)$. G_1 and G_2 are called structural equivalent.*

For showing the equivalence of (G_1) and (G_1), cf. [34], the grammars (G_1) and (G_2) are transformed into grammars $(G_1)'''$ and $(G_2)'''$ in a normal form. The latter grammars only differ in the names of the nonterminals if they are equivalent. Then there exists a isomorphism $N_1''' \to N_2'''$.

Algorithm 3.2 *The transformation of a parenthesis grammar (G) is done by the following steps.*

(1) Construct an invertible equivalent grammar $(G)'$ with $L((G)) = L((G)')$ such that no two right-hand sides of a production are equal.

(2) Construct an grammar $(G)''$ with $L((G)') = L((G)'')$ that does not contain any useless terminal or nonterminal symbol. A terminal symbol is called useless iff it does not occur in a word of $L((G)'')$. A nonterminal symbol X is useless iff it does not produce a terminal word or there is no word $w \in (M \cup N)^*$ including X that can be derived from the start symbol.

(3) Construct a reduced grammar $(G)'''$ with $L((G)'') = L((G)''')$ that has no redundant nonterminal symbols. Two nonterminal symbols are redundant iff both can be replaced by one additional terminal symbol without changing the language.

Finally, compare the grammars $(G_1)'''$ and $(G_2)'''$ for equivalence.

Since parenthesis grammars can be transformed in an unique normal form they can also be checked for inclusion. If every production of $(G_1)'''$ is in $(G_2)'''$ then $L((G_1)) \subseteq L((G_1))$. This leads to the following

Lemma 3.3 *Let (G_1) and (G_2) be the parenthesis versions of context-free grammars G_1 and G_2, respectively. If $(G_1) \subseteq (G_2)$ then $L(G_1) \subseteq L(G_2)$. G_1 is called structural contained in G_2. $(G_1) \subseteq (G_2)$ is decidable.*

In steps (1) and (3), the above algorithm constructions exponential many subsets of nonterminals in the worst case. Additionally, the comparison of the normal forms is exponential because productions may have alternative right hand sides. The algorithm must compare each alternative right hand side of a production of the first grammar with any alternative right hand side of the

corresponding production in the second grammar. It backtracks when it fails and tries the next mapping of alternatives. Hence in general, it cannot run in polynomial time. We therefore define a stronger condition, computable in polynomial time, to guarantee inclusion of context-free languages.

The first simplification is that we assume the grammars to be deterministic. This implies step (1) does not change the grammar and can be omitted. Step (2) is polynomial and is performed as is. Step (3) is omitted hoping that both grammars contain the same redundant nonterminals. The problem is, however, to decide which alternative productions to compare. Our algorithm uses the $SLL(k)$ [32] property of some grammars: A top-down parser is able to choose the right alternative of any production by comparing the next k input tokens with the k first admissible tokens of each alternative right hand side. Obviously the intersection of the k first admissible tokens must be empty for each pair of alternative productions. Additionally, these sets are computable efficiently, cf. [39].

The algorithm for checking structural equivalence is defined by the constructive proof of the following

Theorem 3.4 *Let (G_1) and (G_2) be the parenthesis versions of context-free grammars G_1 and G_2, respectively. If G_1 and G_2 are $SLL(k)$ grammars without (or the same) redundant nonterminals, structural equivalence of G_1 and G_2 can be computed efficiently.*

Proof: $SLL(k)$ grammars are deterministic. For each grammar, we compute the sets of the first k admissible tokens for each alternative of each production. These sets are pairwise disjoint for the alternative right hand sides of a production, otherwise the grammar is not $SLL(k)$. We only have to compare those right hand sides of productions from (G_1) and (G_2) for which the computed sets are equal. Beginning with the start symbol this requires a single top down traversal – backtracking is not necessary. ◇

Although the $SLL(k)$ property of grammars is a sufficient criteria to decide structural equivalence and containment, respectively, in polynomial time, it is often too restrictive. It is sufficient to distinguish alternative productions with the same number of nonterminals by the first k admissible tokens. This heuristic is obviously correct since two productions with different number of nonterminals cannot be structural equivalent.

Example 3 *Consider the grammars of* `stack` *and* `extended_stack`. *The grammars are not $SLL(k)$ for any k. However, algorithm 3.2 can always decide which alternatives should be compared (the rightmost column describes why other com-*

parisons need not be considered):

$$
\begin{array}{rcll}
S & \leftrightarrow & S' & \textit{start symbols} \\
\\
S\ T & \leftrightarrow & S'\ T' & \\
T & \leftrightarrow & T' & \textit{same number of nonterminals} \\
\\
\texttt{push}\ S\ \texttt{pop} & \leftrightarrow & \texttt{push}\ S'\ \texttt{pop} & \\
\texttt{push pop} & \leftrightarrow & \texttt{push pop} & \textit{same first 2 admissible tokens}
\end{array}
$$

The algorithm decides that the grammars are structurally contained since no comparison leads to a contradiction.

The following heuristics relax the criteria of structural containment. Although structural containment is a sufficient criteria, it is still too restrictive as the following example shows.

Example 4 *Consider again our example* `stack`*. A second* `stack'` *equivalent to* `stack` *except for an additional production* $S' \rightarrow S$ *in its grammar.* S' *is the start symbol of* $G($`stack'`$)$*. Both languages are still equivalent. However, the grammars are not structural equivalent because of the additional production. To avoid these kind of problems, we could eliminate chain productions before we check for structural equivalence.*

Let another stack `stack''` *be equivalent to* `stack` *except for the missing production* $S \rightarrow S\ T$ *in the grammar* $G($`stack''`$)$*. Obviously,* $L($`stack''`$) \subseteq$ (`stack`) *since* $G($`stack''`$)$ *is structurally contained in* $G($`stack`$)$*. However, if we eliminate chain productions the nonterminal* T *disappeared. Then the grammars are not structurally contained any longer.*

To avoid these problems we perform the following:

Algorithm 3.5 *Eliminate chain productions. Assume at a certain step in our comparison of the reduced grammars* G *and* G'*, algorithm 3.2 decides that the two production* p *and* p' *are not structurally contained. If the following holds:*

1. $X_0 \rightarrow X_1$ *is an alternative in* p*. The set of the first* k *admissible tokens of* X_1 *is* K*.*
2. $X'_0 \rightarrow \ldots$ *is the production* p'*. The set of the first* k *admissible tokens of* X'_0 *is* K'*.*
3. $K' \subseteq K$*.*

Then substitute p *by* $X_0 \rightarrow X_1$ *in* G*, perform step (2) of the algorithm 3.2, eliminate chain productions and compare the obtained grammar* $\bar{G}$ *with* G'*.*

Since $L(\bar{G}) \subseteq L(G)$, it holds that $L(G') \subseteq L(G)$ if G' is structurally contained in $\bar{G}$. Since algorithm 3.5 removes at least one production from G for each cycle, the algorithm is still polynomial.

We conclude this discussion with the remark that, w.r.t. our application, languages L containing ε need not be distinguished from languages that do not contain ε. ε is a legal prefix of all sentences of any language. In the terminology of our application, it is always legal to call no method of an object. Every $SLL(k)$ grammar G with $L(G) = L \cup \{\varepsilon\}$, can be transformed into a $SLL(k+1)$ grammar G' with $L(G') = L$, cf. [39]. We perform this transformation if necessary.

In addition to the criteria of "structural containment" of context-free languages, we know of sublanguages for which the inclusion problem is efficiently computable. The results are summarized below.

It is decidable whether or not a regular language L is subset of a regular language L'. For every regular language produced by a regular grammar G there exists a (in general nondeterministic) finite automaton A accepting L [9]. For every nondeterministic finite automaton there exists a deterministic one accepting the same language [31]. The construction is exponential. The minimization is polynomial in time [18, 28]. Two minimized finite automata accepting the same language are isomorphic and differ only in the names of the states [29]. The inclusion of two minimized finite automata can be checked in linear time for example by depths first search over the states.

The finite automaton that is constructed from a regular grammar corresponds to the derivation tree of the grammar. If the grammar is deterministic, the automaton is deterministic as well. The problem becomes computable in polynomial time. It easily follows

Theorem 3.6 *If the call sequences of two classes A and A' are defined by deterministic regular grammars, conformance of A and A' can be checked in polynomial time.*

We now look at the inclusion problem for Dyck languages. The Dyck language over an alphabet $T = \{a_i, a_i' | i = 1, \ldots, n\}$ is generated by the grammar $G = (X, T, \{X\}, P)$ where $P = \{X \to XX; X \to a_i X a_i' | i = 1, \ldots, n\}$. A Dyck language is the language over an alphabet containing pairs of parenthesis that are properly set.

Theorem 3.7 *If the call sequences of two classes A and A' are defined by Dyck languages $L(A)$ and $L(A')$, conformance of A and A' can be checked in polynomial time.*

Proof: For Dyck language L and L' it holds that $L \subseteq L'$ iff the alphabet M' of the language L' includes the alphabet M of L: There is exactly one production $X \to a_i X a_i'$ in P for each of the parenthesis a_i, a_i'. It is easy to see that Dyck languages depend only on their alphabet and the inclusion problem can be reduced to a check of the alphabets. This can be done in polynomial time. $\diamond$

This section defined three sufficient criteria computable in polynomial time that imply conformance of two classes A' and A:

1. $G(L(A))$ and $G(L(A'))$ are $SLL(k)$ and $G(L(A))$ is structural contained in $G(L(A'))$, cf. theorem 3.4. Additionally we can relax the $SLL(k)$-

condition such that only productions with an equal number of nonterminals in its alternatives must be distinguished by the first k admissible tokens.

2. $L(A)$ and $L(A')$ are regular languages and $L(A) \subseteq L(A')$, cf. theorem 3.6.

3. $L(A)$ and $L(A')$ are Dyck languages and $L(A) \subseteq L(A')$, cf. theorem 3.7.

In general, it is not decidable whether or not two context-free languages L and L' are regular and Dyck languages, respectively. We omit to describe known necessary criteria.

4 Correct Application

By now we considered whether or not two classes conform to each other. We required a context-free grammar that defines a language of legal sequences of message calls. We did not consider whether or not the call to an object of such a class complies with this language. Unfortunately, this cannot be decided statically in the general case since it would require to solve the aliasing problem, which is intractable in general [20]. However, we can derive at least dynamic consistence checks from the language. These checks would improve the safety of our system in the sense that inappropriate method call would be rejected immediately, i.e. a misplaced method call cannot lead to errors somewhere in other objects.

For distributed systems, the same technique could be used do generate synchronization code. Since this code is generated from a specification, *inheritance anomalies* as described by MATSUOKA and YONEZAWA in [23], cannot occur.

We assume the grammar to be a $(S)LR(k)$, $(S)LL(k)$, or $LALR(k)$ grammar. WAITE and GOOS give a summary of these classes of grammars in [39]. These properties imply that the grammar is deterministic (which restricts the languages we can handle). With the algorithms described e.g. in [39] we could automatically generate stack machines accepting only the correct sequence of calls. There are a lot of tools available which implement these generation algorithms: ell, yacc, bison, lalr, lark are only some examples. GROSCH describes the latter generator and gives an overview over the others [17].

All generated parsers run in time linear to the length of the sequence of method calls. Additionally, they fulfill the requirement that the first misplaced method call is detected and recognized as an error.

Unfortunately, we cannot check the correctness of an application at compile time. However, our next step is to introduce an analysis which allows to remove the runtime check for some applications of a class. For correctly applied objects, cf. definition 2.6, we can omit the runtime check. To determine whether or not objects are applied correctly we have (i) to distinguish different objects at compile time and (ii) to prove that for each single object a, generated as an instance of class A, it holds $L(a) \subseteq L(A)$.

Using the algorithm from the previous section we can test (in a pessimistic way) whether or not a is applied correctly.

First we show how (i) can be approximated. The technique to do this is COUSOT'S *abstract interpretation* [10]. We require a model of the dynamic behavior of a program and interpret the program w.r.t. this model, i.e. we update the model in each step. This is done up to a fix-point. The concrete model that we may apply in our context is the *Storage Shape Graph* (SSG) defined by CHASE, WEGMAN, and ZADECK in [8]. This Storage Shape Graph is computed for each program point. The result is a graph containing variable nodes for each variable, and storage nodes for objects. Edges in this graph represent references, either from variables to objects or from object attributes to other objects.

Obviously, the SSG cannot be exact because not all branches are statically predictable. Therefore an object node in the SSG represents a set of objects that might be created at runtime. Variables that point to only one SSG node are called deterministic variables. An object node unifies all objects that are created at the same point in the program and which are referenced to by the same set of deterministic variables. Similarly we call an attribute deterministic, if it points to only one SSG node.

The original algorithm distinguishes weak and strong updates during interpretation. To identify whether weak or strong updates should be performed, the analysis identifies nodes that correspond to single runtime objects. This is exactly what we require to approximate (i). We compute (ii) for all objects that are abstracted by a single SSG node and that additionally are accessible only via deterministic variables and attributes.

Remark: The results of storage shape analysis can by improved by adding two heuristics: we can distinguish objects if they are created at the same point in the program but if the call path to this creation is different. To maintain computability, we only consider the last k computational steps. This is known as *k-bounded approximation* and was introduced by SCHWARZ in [35, 36]. The second technique that sharpens the shape of the storage is to distinguish the first l elements of lists instead of unifying all non anchor elements [21]. Both heuristics increase the time for the analysis significantly already for small k and l. ◇

The technique to determine (ii) is *program slice*: We filter the program such that it only contains those parts that are of interest for just a single object. This contains all calls to variables or fields pointing to this object. Additionally we have to consider the control structure of the program in order to determine the sequence of these calls. The set of possible sequences is represented by a grammar which we construct while interpreting the program on an abstract level. We assume that our programming language contains conditional and loop statements. It further contains method calls and basic assignments of values to variables.

Algorithm 4.1 *Let a be the object of class A for which we compute the slice. For each single program point, the Storage Shape Graph defines the set of vari-*

ables and attributes pointing to a. We denote this set the access path set *of a at a certain program point. The grammar $G = (S, M_A, N, P)$ is that describes $L(a)$ is initially $N = \{S\}, P = \emptyset$. We associate the start symbol S with the main method, i.e. with the entry point of the program. For each method definition we add a new unique nonterminal symbol to N and associate it with this method.*

For each method, we compute the set of rules describing the sequence of calls to a generated by this method. Let X_m be the nonterminal symbol associated with an arbitrary method m. We add a production $X_m \to X_m^1 \ldots X_m^n$ to P where the X_m^i are determined in the following way:

We traverse m beginning with its entry point in textual order; initially $i = 1$; i is increased whenever an X_m^i is defined (by the $\leftarrow$ operation). One of the following cases may occur:

1. *We skip basic assignments.*
2. *For each call to a method m', $X_m^i \leftarrow X_{m'}$, where $X_{m'}$ is the nonterminal symbol associated with m'. We add a production $X_{m'} \to m'$ to P iff the call is to a variable of attribute from the access path set of a.*
3. *For each loop, $X_m^i \leftarrow Y$ where Y is a new nonterminal symbol. We add Y to N and $Y \to \varepsilon$ to P. Recursively we determine the grammar of the loop body by the same algorithm.*
4. *For each conditional statement, $X_m^i \leftarrow Y$ where Y is a new nonterminal symbol. We add Y to N and $Y \to Y'|Y''$ to P, Y', Y'' are also new nonterminal symbols. Y' determines the true block of the conditional statement, Y' the false block. If there is no false block, we replace Y'' by ε in P. Recursively, we determine the grammar of the true block and the false block (if any) by the same algorithm.*

We eliminate useless terminal and nonterminal symbols. Finally we eliminate chain and ε productions.

Remark: Case statements are treated like cascades of if statements. Polymorph calls are treated like case statements over monomorph calls. ◇

Lemma 4.2 *The grammar G is context-free.*

Proof: For each production in P, there is only a single nonterminal symbol on the left hand side. ◇

Example 5 *We consider an application of the stack class from our previous examples:*

```
class Main is
   s:STACK(INT);
   f(x:INT) is
      if x>0 then
```

```
            s.push(x);
            f(x-1);
            s.pop;
         end; --if
      end; --f
      main_method is
         s:=new(STACK(INT));
         x:=input;
         f(x);
      end; --main_method
   end; --Main
```

There is only one object created from stack. The access path set is empty before the first statement of the `main_method`*. For all other points in the program, this set is* $\{\mathtt{s}\}$*.*

According to algorithm 4.1, the `main_method` *generates the productions:*

$$S \;\rightarrow\; X_{new}\ X_{input}\ X_f,$$

where X_{new}*,* X_{input}*, and* X_f *are nonterminals for the subsequences of calls generated by the methods* `new`*,* `input`*, and* `f`*, resp. We do not consider the former two methods since they do not generate calls to* s*. They are deleted later. The method* f *would create the productions:*

$$\begin{aligned} X_f &\rightarrow X_<\ Y \\ Y &\rightarrow Y' \mid \varepsilon \\ Y' &\rightarrow X_{push}\ X_f\ X_{pop} \\ X_{push} &\rightarrow \mathtt{push} \\ X_{pop} &\rightarrow \mathtt{pop}. \end{aligned}$$

We do not further consider the methods $<$*,* `push` *and* `pop` *for this example. We assume there is no call from these methods to the considered object.*

After the described deletions and the elimination of chain productions we obtain a grammar $G = (X_f, M_{stack}, N, P)$ *with* $N = \{X_f\}, M_{stack} = \{\mathtt{push}, \mathtt{pop}\}$ *and with the following productions:*

$$\begin{aligned} X_f \;\rightarrow\; & \mathtt{push}\ X_f\ \mathtt{pop} \mid \\ & \mathtt{push}\ \mathtt{pop}. \end{aligned}$$

with X_f *as the new start symbol. Obviously* $L(G) \subset L(\mathtt{stack})$*. Easy computations show that the two grammars are* $SLL(2)$ *and* G *is structurally contained in* $G(\mathtt{stack})$*.*

By construction, the grammar G defines a pessimistic approximation of the set of all sequences $L(a)$ of calls to an object a that might occur in a run of the program. More precisely, for each considered object a, algorithm 4.1 computes a grammar G such that $L(a) \subseteq L(G)$ for each execution of the program. This observation leads directly to the following central

Theorem 4.3 *Let G be the grammar computed by algorithm 4.1 for an object a of class A. If $L(G) \subseteq L(A)$ then a is guaranteed to be applied correctly.*

Of course, the runtime check can be omitted in this case. Because of lemma 4.2 we may apply the results from the previous section to check $L(G) \subseteq L(A)$.

5 Conclusion

We extended the definition of conformance of classes by not only considering the interfaces, i.e. the sets of methods, but also the admissible call sequences to these methods. Call sequences are defined by languages over the interfaces, cf. section 2.

We discussed context-free languages, as required for many standard data structures like stacks, buffers, channels etc. We proved several sufficient criteria for conformance that can be checked statically and in an efficient way, cf. section 3.

We further showed how a dynamic check can be generated from the specification of legal call sequences. These checks guarantee that no method is executed if the sequence of calls is not admissible. This technique can be used to detect errors at the point of their occurrence instead of the point of their symptoms' occurrence. For languages supporting active objects, it can be used to generate synchronization code. This avoids inheritance anomalies. Additionally, we showed an analysis which allows to omit dynamic tests. For certain applications it is guaranteed at compile time that all call sequences are legal, cf. section 4.

The next step is to demonstrate the benefits of our approach for practical problems. Therefore we currently analyze *Karla* – the library of algorithms and data structures[2] developed at our institute. Initial investigations showed that most of the about 250 classes in Karla require at least a regular restriction since most of them must be initialized before being used.

In this work, we only considered criteria that can be checked in polynomial time. From the generation of scanners, we know that some exponential problems can be handled in practice. E.g. the construction of a deterministic automaton from a nondeterministic one is practically harmless. We should find out whether some of our exponential decision problems are practically not serious in our applications.

However, quite a few theoretic questions remain unsolved:

- How can we handle calls from methods back to the system? Our idea is to extend the solutions discussed for acceptors of languages to language transducers. It is an open problem which of the criteria are then decidable and which are efficiently computable.

- How can we specify the sequence of methods that should have been called to arguments and will be called to results, respectively? If the legal call sequences to arguments can be described by regular languages, we simply

[2] http://i44www.info.uni-karlsruhe.de/~ karla

specify the required state. For context-free languages, again a grammar could describe these required call sequences. What are then the implications for the correctness and conformance criteria?

- Can we define better criteria for checking inclusion of context-free languages in polynomial time? Right now we have a sufficient criteria. Is this criteria too strong for some practically relevant cases? Are there more precise criteria computable in polynomial time?

- Can we give better criteria for conformance such that systems are correct by construction? Is it possible to specify the dynamic behavior of objects even more precisely such that we can statically check whether or not these objects can be combined in a reliable way?

- Can we relax the definition of conformance in a way such that a correct but more flexible combination of objects is possible? It can be observed that in concrete applications not always all methods of an object are called but only a few of them. This means that conformance can be considered w.r.t. a used set of methods. Especially, this approach permits proofs of correctness for specialized (partial conforming [13]) classes.

Although there are open problems, our results can be used to improve the safety of object oriented systems and to increase the number of classes that can be reused in a safe way.

References

[1] P. America. Designing an object-oriented language with behavioural subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, volume 489 of *Lecture Notes in Computer Science*, pages 60 – 90. Springer-Verlag, 1991.

[2] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

[3] L. Cardelli. A semantics of multiple inheritance. *Info. and Computation*, 76:138–164, 1988.

[4] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Report 31, August 25, 1988, DEC System Research Center, Palo Alto, 1988.

[5] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report (revised). Technical Report 52, DEC System Research Center, Palo Alto, 1989.

[6] C. Chambers. Predicate Classes. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 268–296, Kaiserslautern, Germany, jul 1993. Springer-Verlag.

[7] C. Chambers. The cecil language – specification and rationale. Technical report, Dept. of Comp. Science and Engeneering – University of Seatle (Washington), december 1995.

[8] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. Technical Report CS-90-06, Department of Computer Science, Brown University, March 1990. Sun, 13 Jul 1997 18:30:14 GMT.

[9] N. Chomsky and G. A. Miller. Finite state languages. *Inf. and Controll*, 1(2):21–112, 1958.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 238 – 252. ACM SIGACT-SIGPLAN, 1977.

[11] O. J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula 67, Common Base Language*. Norway Computer Center, Oslo, 1968.

[12] A. Frick, W. Zimmer, and W. Zimmermann. On the design of reliable libraries. In *TOOLS 17 – Technology of Object-Oriented Programming*, pages 13–23, 1995.

[13] J. Frigo, R. Neumann, and W. Zimmermann. Generation of robust class hierarchies. In *TOOLS 23 – Technology of Object-Oriented Languages and Systems*, 1997.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software Components*. Addison-Wesley, 1994.

[15] G. Goos. Sather-K – the language. *Software – Concepts and Tools*, 18:91–109, 1997.

[16] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison Wesley, 1997.

[17] J. Grosch. Lark – a LR(1) parsergenerator with backtracking. Technical Report 32, GMD Forschungsstelle Karlsruhe, Nov. 1994.

[18] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 3-4(2):161–190 and 275–303, 275 1959.

[19] S.C. Klenee. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princton Univ. Press, 1956.

[20] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[21] James Richard Larus. Restructuring symbolic programs for concurrent execution on multiprocessors. Technical Report CSD-89-502, University of California, Berkeley, 1989.

[22] B. Liskov and j.M. Wing. A new definition of the subtype relation. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 118 – 141. Springer-Verlag, jul 1993.

[23] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Aghga, A. Yonezawa, and P. Wegner, editors, *Research Directionsin Concurrent Object-Oriented Programming*, pages 97–106. MIT Press, 1993.

[24] R. McNaughton. Parenthesis grammars. *J. Assoc. Comput. Mach.*, 14:490 – 500, 1967.

[25] B. Meyer. Applying "Design by Contract". *IEEE Computing (Special Issue on Inheritance & Classification)*, 25(10), october 1992.

[26] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[27] R. Mitchell, I. Maung, J. Howse, and T. Heathcote. Checking software contracts. In R. Ege, M. Singh, and B. Meyer, editors, *TOOLS 17 — Technology of Object-Oriented Programming*, pages 97–106. Prentice Hall, August 1995.

[28] E. F. Moore. *Gedanken experiments on sequential machines*. Princeton University Press, New Jersey, 1956.

[29] A. Nerode. Thourghts on a Larch/ML and a new application of LP. In *Proc. Amer. Math. Soc.,*, pages 541–544, 1958.

[30] O. Nierstrasz. Regular types for active objects. In *Proceedings OOPSLA'93*, pages 1 – 15. ACM, 1993.

[31] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res.*, 3(2):115 – 125, 1959.

[32] D.J. Rosenkrantz and R.E. Stearns. Properties of deterministic top-down grammars. *Inf. and Controll*, 17:226–252, 1970.

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

[34] A. Salomaa. *Formal Languages*. Academic Press, New York, San Francisco, London, 1973.

[35] J.T. Schwartz. Optimization of very high level languages – I: Value transmission and its corollaries. *Computer Languages*, 1:161 – 194, 1975.

[36] J.T. Schwartz. Optimization of very high level languages – II: Deducing relationships of inclusion and membership. *Computer Languages*, 1:197 – 218, 1975.

[37] D. Stoutamire. *The pSather1.0 Manual*. International Computer Science Institute, 1995.

[38] B. Stroustrup, editor. *The C++ Programming Language*. Addison Wesley, second edition, 93.

[39] W. Waite and G. Goos. *Compiler Construction*. Texts and Monographs in Computer Science. Springer, 1985.

[40] P. Wegner and S.B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proceedings ECOOP'88*, volume 322 of *Lecture Notes in Computer Science*, pages 55 – 77. Springer-Verlag, 1988.

[41] J.M. Wing, E. Rollins, and A.M. Zaremski. Thourghts on a Larch/ML and a new application of LP. In U. Martin and J.M. Wing, editors, *1st Internat. Workshop on Larch*. Springer, 1993.

[42] A.M. Zaremski and J.M. Wing. Specification matching of software components. *ACM Trans. Software Engineering and Methodology.*, 6(4):333 – 369, 1997.