Construction of Robust Class Hierarchies

A. FRICK
Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, Am Zirkel 2, Postfach 69 80,
D-76128 Karlsruhe, Germany

G. GOOS
Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, Am Zirkel 2, Postfach 69 80,
D-76128 Karlsruhe, Germany

R. NEUMANN
Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, Am Zirkel 2, Postfach 69 80,
D-76128 Karlsruhe, Germany

W. ZIMMERMANN
Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, Am Zirkel 2, Postfach 69 80,
D-76128 Karlsruhe, Germany

SUMMARY
This article discusses the problem of constructing robust class libraries. Further design criteria include the flexibility of class libraries, the efficiency of the implementations, and their safe extensibility. We show that it is possible to design robust libraries to satisfy any two of the requirements at the same time. Although the solution may require an exponential growth in the number of classes compared to the original design, this apparent class explosion can be controlled by generating only the necessary additional classes automatically. As an application demonstrating both the theoretical problems and the power of our generator approach, the design of a library modeling data structures and algorithms for graphs is considered. Both the discussion and the results in this article generalize to other domains.

INTRODUCTION
A major promise of object-oriented programming has traditionally been increased reusability. The concept of inheritance was claimed to offer more possibilities for reusing components than, for instance, modular languages. However, only little research exists on the systematic and large-scale design of class hierarchies 1-3. This article strives to bridge the gap by studying a reason for the combination of design criteria and their respective tradeoffs. Particular attention is given to the criteria robustness and correctness. A library is called robust if (1) no errors can be introduced into an application due to its use or inheritance from it, (2) error messages do not point deep into library code but into the user’s application, and (3) constraints on usage of the library that can be expressed as types are indeed represented by types. Then, the violation of these constraints can be detected at compile time. If these constraints are not represented by types, they cannot be checked by compilers.

This article demonstrates that the inheritance concepts of existing programming languages (and therefore, libraries built using inheritance and polymorphism) do not support robustness well, which may result in unexpected errors in both libraries and applications built on top of them. We demonstrate these kinds of errors and show how to make class hierarchies based on inheritance 4-6 robust. Besides robustness, the following properties of class libraries are desirable as well:

Flexibility: This property covers the requirements on usage contexts of classes. In particular, polymorphic classes can be used in more contexts than monomorphic classes. Furthermore, it is desirable if implementations of data structures and algorithms can be changed at runtime.

Efficiency: This property addresses whether it is possible to implement abstract classes and their methods by the most efficient known data structures and algorithms. For example, the class String in the Java base libraries is an immutable class, i.e. a class with value semantics. Whenever such an object of class String needs to be modified, a new object with the modified values has to be created. This especially means that even exchanging a single character by another cannot be done in time O(1), since first the complete string has to be copied.

Safe Extensibility: An extension to a library must not invalidate programs that use a version of the library before the extension. In particular, extensions must not change the semantics of existing methods w.r.t. their use, or remove methods from a class.

A more formal definition is given in BASIC DEFINITIONS. For simplicity reasons, we assume that a library consists of abstract classes defining interfaces and implementation classes implementing these interfaces. This is not a real limitation. Assume that a library does not have the desired structure. Then, for every concrete class C, an abstract class AC containing just the interfaces of C can be added, and C can be modified such that it inherits the interface class AC. For now, we regard abstract classes as abstract data types and implementation classes as data structures implementing abstract classes. This view is refined in BASIC DEFINITIONS as well.

Ideally, robust class libraries should satisfy all the above non-functional properties. Unfortunately, there are trade-offs between these: Often – especially when developing long-running applications – implementations of data structures or algorithms need to be exchanged in order to achieve maximum efficiency. Accordingly, several design patterns 6 have been developed to capture different aspects of this behavior, e.g. Bridge, Strategy or State. A robust and efficient class library must maintain consistency between algorithms and data representations, if implementations of data structures are allowed to be exchanged by the user at runtime. Forcing the library user to guarantee consistency makes a library hard to use, as violations of this rule may lead to inconsistencies, which in turn may cause an accidental misuse of the library. A robust class library therefore either cannot allow the use of efficient algorithms relying on a particular data structure, or they cannot allow the user to change implementations of data structures at runtime, therefore not offering the maximal desirable flexibility.

Throughout the remainder of this article, the domain of graphs serves as a rich source of examples. It is also a well-known domain that demonstrates both the problems and the power of the generator approach. It should be noted, however, that our results generalize to other domains as well. The presentation of the material is as follows: Section BASIC DEFINITIONS reviews the basic object-oriented terminology and defines the notions of correctness, robustness, flexibility, and safe extensibility more precisely. Section INHERITANCE ANOMALIES demonstrates how errors can be introduced by inheritance, even if every class by itself is correct. The analysis

29/10/1999 11:16 PAGE PROOFS
results in a precise definition of several kinds of inheritance. Section GENERICITY ANOMALIES extends the discussion to parameterized (a.k.a. generic) classes. Section INHERITANCE HIERARCHIES provides an approach to construct robust inheritance hierarchies satisfying the individual non-functional properties and discusses the trade-offs between these properties in more detail. Section GENERATION OF CLASS HIERARCHIES introduces our generator-based methods to automatically generate class hierarchies from small specifications. Although the number of abstract classes may grow exponentially \(^6\), which makes a manual implementation of the solution impractical \(^7\), the good news is that every solution can be generated from the same comparatively small specification \(^8\), and that only those classes that are actually required by the library user must be generated. Therefore, an interactive user-interface should be provided for supporting the library user to identify and generate the required classes makes our approach feasible. In RELATED WORK, we study how well existing software libraries satisfy the above criteria, and highlight other systematic approaches to the construction of robust object-oriented class libraries. The CONCLUSIONS present our vision on the architecture of robust libraries and their usage. In particular, we discuss how a library user can customize his own classes using the generator techniques introduced in Section GENERATION OF CLASS HIERARCHIES.

### BASIC DEFINITIONS

We first define basic terminology in object-oriented design, programming and languages. Then we define the notion of correctness of classes and libraries. Finally, we define the non-functional properties.

Our methodology can be applied to any typed object-oriented language, preferably with genericity. It also applies to untyped object-oriented languages, except that some statically checkable properties can be checked only at runtime in untyped object-oriented languages.

#### Object-Oriented Terminology

This subsection defines the notion of classes, types, attributes, objects, state, methods, inheritance, genericity, and polymorphism. A class is a collection of methods and attributes and has a name. Each class defines a type which is identified with its name. An attribute is pair of names and types, denoted by name : type. A method is a procedure or function and has a name and a signature. A signature is pair where the first component consists of a list of types and the second component consists of a type. A method is denoted by \(\text{name} : \text{type1} \times \cdots \times \text{type}\), → \(\text{type}\). There is a special type VOID which refers to the type defined by the class where it is used. We will denote classes as follows:

1. **class** name lib
2. **attributes**
3. **methods** methods
4. **end**

Within the scope of a class \(A\), the attributes play the role of variables. If \(x : T\) is an attribute and \(T\) is a basic type, then \(x\) contains a value of type \(T\). Otherwise, it contains a reference to objects of type \(T\). Let \(A\) be a class with attributes \(a_1 : T_1, \ldots, a_n : T_n\). An object \(o\) of class or type \(A\) is a record with components \(a_1, \ldots, a_n\), where \(a_i\) contains a value of type \(T_i\) if \(T_i\) is a basic type, and a reference to an object of type \(T_i\) otherwise. The state of an object \(o\) comprises the values and references of its components together with the states of the referenced objects. If \(x : T\) is an attribute and \(T\) is a type with attribute \(y : T\), then \(x.y\) denotes the reference or value denoted by the \(y\)-attribute of \(T\) of the object referenced by \(x\). This notation is extended inductively in the straightforward way. The operation \(\#A\) creates a new object and returns a reference to it. The equality of objects is based on their reference, i.e. on object identities. However, sometimes structural equality is used. This notion is based on the fact that objects can be viewed as vertices of a directed graph and references as edges (e.g. see \(^9,10\)). Two objects are structurally equal iff their directed graphs are isomorphic.

The interface of a method declares the signature with its parameters. \(m(x_1 : T_1, \ldots, x_k : T_k) : T\) denotes a method \(m\) with signature \(m : T_1 \times \cdots \times T_k \rightarrow T\) where \(x_i\) is a variable of type \(T_i, i = 1, \ldots, k\). \(T = \text{VOID}\) if \(m\) is a procedure. Methods may have an implementation in a particular programming language. In our case, we also consider methods without implementations. These methods are called abstract. An abstract class is a class that contains abstract methods. Classes without abstract methods are called implementation classes. In our article, abstract classes are denoted by the keyword abstract.

In our examples, we use the language Sather-K \(^11\) for implementing methods. We just use the usual control structures of while-loops, for-loops, and conditionals with their usual semantics \(^9\). The \(\text{lhs} := \text{rhs}\) operator denotes an assignment. It means that if \(\text{lhs}\) is an object of a basic type, then the value denoted by \(\text{lhs}\) will be the content of \(\text{rhs}\). Otherwise \(\text{rhs}\) denotes a reference to an object which is copied to \(\text{lhs}\). Within the implementation of a function, \(\text{rhs}\) denotes the value/reference to be returned. Implementations of methods may declare local variables. Such a declaration is denoted also by \(z : U\) and has the same meaning as attribute declarations, i.e. it is a variable containing the values of \(U\) if \(U\) is a basic type and a reference to objects of type \(U\) otherwise.

Let \(m(x_1 : T_1, \ldots, x_k : T_k) : T\) be the interface of a defined in a class \(A\) and \(r\) be a reference to an object of type \(A\). Furthermore, let \(t_1, \ldots, t_n\) denote objects of type \(T_1, \ldots, T_k\), respectively. Then \(x.m(t_1, \ldots, t_n)\) denotes a function call of the object designated by \(x\). Its meaning is the following: First the values/references defined by \(t_i\) are copied to \(x_i, i = 1, \ldots, n\), then the body of \(m\) is executed. After termination, the function call denotes the value/reference contained in \(r\). Within the method, \text{self} is the reference to the object on which \(m\) is executed. This reference cannot be changed explicitly, e.g. assignments to \text{self} are forbidden. If \(m\) is a procedure, then the procedure call does not denote a value or a reference and cannot be used in expressions. The function/procedure call has a side-effect on \(x\) or \(x_i, i = 1, \ldots, n\), if its execution changes the state of \(x\) or \(x_i\), respectively. We assume that for each class there is a special procedure \text{init} that performs initializations of objects. This method must not be called explicitly, but is instead automatically executed upon creation of an object. If this procedure has parameters, then the object creation has arguments. E.g. suppose a class \(A\) has the initialization procedure \text{init}(x) : \text{VOID}\). Then, the creation of an object of class \(A\) must have one argument which must be an integer, e.g. \#A(2) creates an object of class \(A\).

A generic or parameterized class is a class with the form

\[\text{class name lib attributes methods end}\]

\footnote{This might be a reason why most existing libraries focus on only one non-functional property.}

\footnote{For-loops do not exist in Sather-K. Instead streams are used. For simplicity, we use for-loops.}

\footnote{Template classes in C++ are examples of parameterized classes.}
In this subsection, we define a notion of correctness based on class invariants and method pre- and postconditions. These assertions are formulas of first-order predicate logic. They use the definitions of the previous subsection. A HOARE-triple is denoted by $[\phi_1 \land \phi_2]$ where $\phi_1$ and $\phi_2$ are first-order formulas and $s$ is a statement. The HOARE-triple has the following meaning: if before execution of $s$, $\phi_1$ is satisfied, then after execution of $s$, $\phi_2$ is satisfied. The notion of free and bound variables is as usual. $\phi[y/x]$ denotes the formula $\phi$ where the free variable $y$ is replaced by $x$.

The symbol $\equiv$ denotes the equality of values of basic types and on object references; the symbol $\equiv$ denotes structural equality of objects.

A class invariant $Inv$ of a class $A$ is a first-order formula that can use the attributes and methods of class $A$. A precondition $Pre_{m,A}$ of method $m$ of class $A$ is a first-order formula that can use attributes, methods of $A$, and parameters of $m$. Similarly a postcondition $Post_{m,A}$ of method $m$ of class $A$ is a first-order formula that can use attributes, methods of $A$, and parameters of $m$. It can also use the variable $res$, if $m$ is a function, and the variables $x'$ if $x'$ is an attribute of $A$ or parameter of $m$. $x'$ denotes the value or content of $x$ before the execution of $m$. $Self$ denotes the state of the object referenced by $Self$ before execution of the method $m$. To simplify the discussion, only side-effect free methods, constructors and destructors are allowed as part of an invariant, precondition or postcondition.

A class $A$ is locally correct if the following two conditions are satisfied:

1. $[Pre_{m,A} \land \#A(x_1, \ldots, x_k) \land Post_{m,A} \land Inv_A]$
2. For each method $m(x_1, \ldots, x_k : T_j) : T$ of $A$ with body $s$:
   $[Pre_{m,A} \land Inv_A \mid s (Post_{m,A} \land Inv_A)]$

A class $A$ is used correctly iff

1. for every method creation $Pre_{m,A}$ is satisfied and
2. for every object $x$ of class $A$ and every method call $x.m(t_1, \ldots, t_k)$ the precondition $Pre_{m,A}$ is satisfied on the object referenced by $x$ with the arguments $t_1, \ldots, t_k$.

These definitions follow the “Design-by-Contract”-principle: the user of a class must ensure the preconditions, the class must ensure invariants and postconditions. Observe that (i)-(iv) do not require $Inv_A \Rightarrow Pre_{m,A}$ or $Post_{m,A} \Rightarrow Inv_A$. The advantage is that the user of a class need not to ensure the invariant when it is separated from pre- and postconditions.

Local correctness of a library does not imply that no runtime-errors may occur, as inheritance, polymorphism and genericity introduce complex interactions between classes (cf. Section INHERITANCE ANOMALIES and GENERICITY ANOMALIES).
Non-Functional Properties

This subsection gives a precise definition of the non-functional properties discussed in the INTRODUCTION. The definitions are based on class specifications. Some method specifications (e.g., operations on Java strings) may prohibit their implementations to have side-effects, which we denote by a predicate unchanged on references to objects. The meaning of unchanged(o) is that no object is changed that can be referenced directly or indirectly by o and the reference o is not changed. Formally, unchanged(o) is an abbreviation for

\[ d' = o \land \alpha \equiv d', \]

The negation of the predicate in (1) is called modified. \( \text{unchanged}(\text{self}) \) is abbreviated by unchanged. Side-effect free methods on self are called queries, the others modifiers. If each method of a class is a query, then a class is called immutable. In particular, all immutable classes have value semantics, i.e. an object state cannot be changed after the initialization of that object. Observe that this definition applies to abstract classes as well as to implementation classes. Mutable methods need not cause side-effects on self. Their specification just do not exclude the possibility that an implementation has side-effects.

A library is called robust if every usage of a library class within the library is correct according to (iii) and (iv). The notion of flexibility of a library is more difficult to define as for example robustness. As a relative, rather than an absolute criterion, we can define that a library \( L_1 \) is more flexible than a library \( L_2 \) if \( L_1 \) provides at least the same functionality as \( L_2 \). For instance, the possibility of dynamically exchanging implementations for data structures and algorithms provides more functionality than otherwise. Polymorphic class hierarchies provide more functionality than non-polymorphic class hierarchies since the subtypes can be used instead of their supertypes which would not be possible if the classes are not in a subtype hierarchy. Also, extending a library may add more flexibility since more classes or methods can be used.

Efficiency of a library is the presence and the possibility of implementing all known efficient algorithms and data structures. It has been shown for example that it is impossible to implement some algorithms without side-effects (e.g. array updates that take constant time). Hence, e.g. immutable classes do not guarantee that every algorithm can be implemented. In this article we discuss mutability vs. immutability of classes instead of efficiency.

We conclude this subsection with a small discussion of design philosophy for libraries. It is desirable to have well-defined rules to design a library. These rules should define and require a coherent library structure. Design rules not only contain coding rules but also rules how to design inheritance hierarchies, when to use abstract classes etc. A particular example of a design rules is that the functionality of a data type is modeled by an abstract class \( C \) and each implementation of \( C \) inherits from \( C \). Section INHERITANCE HIERARCHIES contains examples of design rules for inheritance hierarchies. As the example of immutable classes has indicated, certain design choices may result in an overall loss of efficiency. Safe extensibility is yet another property rooted in the library structure. We only consider library extensions which do not violate the design rules. In this sense, a library \( L \) is safely extensible if for every program \( \pi \) using \( L \) without compile time and runtime errors there is no library extension \( L' \) of \( L \) satisfying the design rules of \( L \) such that \( \pi \) using \( L' \) instead of \( L \) does not change the semantics of \( \pi \). Sometimes, such semantic changes may lead to compile time or run time errors. Safe extensibility of a library (resp. its design rules) prevents such semantic changes.

INHERITANCE ANOMALIES

This section summarizes the results on inheritance of \( ^{4,5} \). The combination of inheritance and polymorphism may introduce runtime errors, and global system correctness cannot necessarily be inferred from the local class correctness, as the following example indicates:

Example 1: Consider the class \( \text{DG}(\text{VERTEX}, \text{EDGE}) \) of directed graphs. For example this class contains a method \( \text{connect}(v_1, v_2 : \text{VERTEX}) \) that adds the edge \((v_1, v_2)\) to a directed graph. \( \text{connect} \) is a modifier. The class of acyclic directed graphs, \( \text{ADG}(\text{VERTEX}, \text{EDGE}) \), contains an additional method \( \text{topsort} \) that topologically sorts an acyclic directed graph. Thus, \( \text{ADG} \) inherits from \( \text{DG} \). Polymorphism means that \( \text{ADG} \) can be used always instead of \( \text{DG} \). However, the modifier \( \text{connect} \) must keep the acyclicity property which requires a stronger precondition on the argument of \( \text{connect} \). Thus, if a \( \text{DG} \)-variable refers polymorphically to and \( \text{ADG} \)-object, the precondition of \( \text{connect} \) is violated although (iii) and (iv) are satisfied.

A class \( B \) inherits a class \( A \) iff \( \text{Intr}_B \supseteq \text{Intr}_A \) and each method \( m \) of \( A \) is also in \( B \). Four kinds of inheritance can be distinguished (cf. Table I and Fig. 1).

<table>
<thead>
<tr>
<th>Table I. Inheritance relations.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conforms to</strong></td>
</tr>
<tr>
<td><strong>Covariant to</strong></td>
</tr>
<tr>
<td><strong>Contravariant to</strong></td>
</tr>
<tr>
<td><strong>Specializes</strong></td>
</tr>
</tbody>
</table>

This section can be used to explain the problem in Example 1, which is an example of a specialization relationship. However, only the conformance relationship retains robustness.
under polymorphism. Example 1 demonstrates the consequences when specializations are used under polymorphism. A conformance relationship guarantees robust subtyping. However, specialization and covariance also occur in practice, as demonstrated by the Example 1, where we have $Pr_{\text{connect},ADG} \supseteq Pr_{\text{connect},DG}$ and $Pr_{\text{dir},ADG} \supseteq Pr_{\text{dir},DG}$. The other inheritance relations still allow for code reuse and building hierarchies, while they cannot be used in a polymorphic context. For instance, many implementations of methods for directed graphs can be reused without modification (i.e., with the same pre- and postconditions) in acyclic directed graphs.

At both the syntactic and semantic levels, most existing object-oriented programming languages do not distinguish between the different types of inheritance discussed above and therefore allow for inheritance anomalies. It remains the task of the designer of a robust class library to design hierarchies that avoid anomalies. Inheriting classes must specify the necessary inheritance relation and the corresponding requirements on invariants, pre- and postconditions in addition to those predicates stemming from the “Design-by-Contract”-principle.

**Remark:** JAVA® has three kinds of inheritances: inheriting from a concrete class, implementing an interface, and extending an interface. However, this distinction does not solve the above problems. In fact, as it can be seen from Example 1, on a syntactic level the interface of ADG could be viewed as an interface extension of the interface of DG. The polymorphism rules of JAVA would imply that any implementation of ADG is also an implementation of DG, causing the same problem as described in Example 1. Hence, the different inheritance mechanisms in JAVA do not solve the problem.

**GENERICITY ANOMALIES**

This section summarizes the results on genericity of 4. We demonstrate that errors can be introduced due to the use of generic classes and give sufficient conditions for avoiding these errors.

**Example 2:** Consider the class $\text{SET}(T)$ defining sets over a universe $T$. This class may have a method $\text{max} : T \to \text{SET}(T)$ computing the maximum of a non-empty set, i.e., $\text{Pre}_{\text{max},\text{SET}} = \text{is}\_\text{empty}$ and

$$\text{Post}_{\text{max},\text{SET}} = \text{unchanged}[\text{self}] \land \text{res} \in \text{self} \land \forall z : T : z \in \text{self} \Rightarrow z \leq \text{res}.$$  

Consider now the class $\text{DEDGE}[V]$.

| (1) | class $\text{DEDGE}[V]$ is |
| (2) | src $\in V$ |
| (3) | dest $\in V$ |
| (4) | $\text{src} = \text{dest} \Rightarrow \text{BOOL}$ |
| (5) | $\text{pre} = \text{true}$ |
| (6) | $\text{post} = \text{true}$ |
| (7) | $\text{self} = \text{src} = \text{dest} \land \text{src} = \text{dest} = \text{res} = \text{e} > 0$ |
| (8) | $\text{end};$ |

Let $s : \text{SET}(\text{DEDGE}[\text{INT}])$ be an object. The call $s.\text{max}$ will fail since it expects that the objects of type $\text{DEDGE}[\text{INT}]$ would have a total order.

**Remark:** If in a programming language genericity is defined by pure syntactic replacement (e.g., C++) then compilers recognize the error at the program points where $\leq$ is used. Consequently, the error message produced by the compiler points into library code, which should be avoided.

The above problem can be avoided by adding a requirement that the instances for the generic parameter of $\text{SET}$ must provide a total order. The restriction of instances of generic parameters

**INHERITANCE HIERARCHIES**

This section discusses construction principles for inheritance hierarchies used in robust class libraries. Subsection DESIGN OF SPECIALIZATION HIERARCHIES discusses different approaches for designing specialization hierarchies. Subsection CONFORMANCE HIERARCHIES OF GENERIC CLASSES discusses the construction of conformance hierarchies of generic classes. The above inheritance hierarchies are hierarchies of abstract classes. Subsection IMPLEMENTATIONS discusses how implementations of abstract classes can be organized robustly, but unfortunately not flexible enough. Subsection IMPLEMENTATION HIERARCHIES introduces a flexible solution at the cost of only allowing representation-independent implementations.

**Design of Specialization Hierarchies**

In the following sections, we discuss four different design alternatives for specialization hierarchies. The first design alternative models specializations as classes that cannot be used polymorphically – in this case inheritance means simple code reuse. The second alternative is called bounded genericity. Bounded genericity must allow to put the following requirements on instances of generic parameters: an invariant $\text{inv}$ that must be satisfied and the presence of methods $\text{method}$ which can be called with preconditions $\text{Pre}_{\text{method}}$ and ensure postconditions $\text{Post}_{\text{method}}$. Thus, the requirements on generic parameters can be expressed as formal specifications of classes. A generic parameter with requirement class $A$ can only be instantiated with types that syntactically conform to $A$. We denote such a restriction of a generic parameter with $T < A$.

**Example 3:** The class $\text{SET}(T)$ of the above example is replaced by $\text{ORDERED}\_\text{SET}(T < \text{ORDERED}(T))$ where the requirement class $\text{ORDERED}(T)$ is defined as follows:

1. abstract class $\text{ORDERED}(T)$ is
2. $\text{inv} : y \in T \Rightarrow x \leq y \leq y \leq x$
3. $\forall x : T : x \leq x$
4. $\forall x, y : T : x \leq y \land y \leq z \Rightarrow x \leq z$
5. $\forall x, y, z : T : x \leq y \land y \leq z \leq x \Rightarrow x \leq z$
6. $\text{method} : (x : T) \Rightarrow \text{BOOL}$ is abstract
7. $\text{pre} = \text{true}$
8. $\text{post} = \text{true}$
9. $\text{end};$

The class $\text{INT}$ syntactically conforms to the type bound $\text{ORDERED}\_\text{INT}$ but class $\text{DGEDE}[\text{INT}]$ does not conform to the type bound $\text{ORDERED}\_\text{DGEDE}(\text{INT})$. **Remark:** At compile time, only the signatures of the instances of generic parameters can be checked. Languages like HASKELL 18, EIFFEL 16, SATHER-K 11, PIZZA 17, and GENERICJAVA 18 actually perform these checks. Since the requirements for a robust library are more stringent, the remaining requirements on the invariants, pre- and postconditions stemming from genericity have to be added to the Design-by-Contract principle.

We extend now the Design-by-Contract principle to generic parameters. In the body of a generic class $A(X < T)$ we require that every object $o$ of parameter $X$ is used correctly w.r.t. to the type bound $T$. If a class $Y$ conforms to $T$ (without necessarily being a subclass), then every contract of $Y$ is also a contract of the type bound $T$. Thus, there are no violations of correct uses in $A(Y)$ due to the instance $Y$ for generic parameter $X$. Therefore, if $A(T)$ is locally correct and $Y$ conforms to $T$, then also $A(Y)$ is locally correct. For this reason, we require that every instance of a generic parameter $X$ must conform to the bound of $X$ (with necessarily being a subclass).
converts the specialization hierarchy into a conformance hierarchy of immutable classes. This solution is safely extensible and flexible but inefficient since it imposes a value semantics on the classes. The third solution is a hierarchy of factorized conforming classes. This approach leads to a flexible and efficient robust hierarchy, which, however, lacks safe extensibility. The last alternative models specializations as dynamic properties of objects, i.e. as specialized states, merging a complete specialization hierarchy into one single class. The problem with this approach is, that properties are checked only at runtime, which means loss of robustness and performance.

For demonstrating the different design alternatives, we use the example of directed graphs, specialized classes with its invariants, illustrate some methods with its preconditions, and provide a specialization hierarchy.

Example 4: We consider again class DG. This class has e.g. the following predicates that check structural properties:

- **has_path(v1, v2)** returns true iff there is a path from v1 to v2.
- **is_art(v)** returns true iff vertex v is an articulation point (i.e. if this vertex is removed, the graph has at least one more weakly connected component).
- **is_bridge(v)** returns true iff vertex v is a bridge (i.e. if this edge is removed, the graph has at least one more weakly connected component).
- **is_acyclic** returns true iff the graph has no cycle.
- **is_free** returns true iff the graph is weakly connected (i.e. connected if the direction of the edges are ignored).
- **DG** has the following modifiers (except):

  - **init** creates an empty graph.
  - **make(V : SET(VERTEX), EE : SET(EDGE))** creates a graph with the set of vertices VV and the set of edges EE.
  - **add_vertex(v)** adds a new vertex v.
  - **connect(v1, v2)** adds an edge from vertex v1 to vertex v2. If v1 or v2 are not present, they are also added.
  - **del_vertex(v)** deletes the vertex v.
  - **del_edge(e)** deletes the edge e.
  - **sym_close** makes the graph symmetric, i.e. adds an edge (v; u) iff (u; v) ∈ E.
  - **trans_close** makes the graph transitive, i.e. adds an edge (v; u; w) iff there is a path from u to w.

Finally, DG has the following methods (except):

- The methods V and E return the set of vertices and edges, respectively.
- The methods idf(v) and odg(v) compute the indegree and outdegree of vertex v.
- The method adj_edges(v) computes the set of all edges that are adjacent to vertex v.
- The method wcc computes the set of all weakly connected components, i.e. all connected components if the direction of edges are ignored.

The predicates **is_acyclic**, **is_free**, **is_weakly_connected** can be used to strengthen the invariant of directed graphs, \(\text{INVDG} = E \subseteq V \times V\). Table II contains the invariants of these classes. The queries **topsort** which computes a topological order of a directed graph and height which computes the height of an acyclic directed graph are only present if the graph is acyclic, i.e. in the classes ADG, TREE, and AWCDG. Table III contains the preconditions of these methods in every class: the rows are the methods, the columns contain the classes, and an entry in row m and column A defines the precondition of method m in class A. E.g. the entry in row connect(v1, v2) and column ADG specifies that connecting v1 and v2 by an edge requires that at least one of the vertices v1 or v2 is not yet present or there is no path from v2 to v1. If this was not true then after insertion of the edge (v1, v2), a cycle would be created. The preconditions of **init** need some explanation. Consider e.g. the precondition

\[
\text{Pre}_\text{vis,ADG}=\exists g : \text{ADG} : g \cap V = V \land g \cap EE = E. \tag{19}\]

\(A = B\) is a defining equality, i.e. \(A\) is an abbreviation for \(B\).

It tells that there is an acyclic directed graph with the same set of vertices as \(V\) and the same set of edges \(E\). Hence, the graph \((V, E)\) satisfies \(E \subseteq V \times V\) and is acyclic.

Table IV illustrates the postconditions. The postconditions are for every graph class the same. For all queries, it is explicitly stated they are side-effect free on the object where they are called (unmodified). The role of the queries \(V\) and \(E\) is implicitly specified in the postconditions of the modifiers. **has_path** is specified using the predicate itself. This is well-defined because the formula

\[
\text{has_path}(v1, v2) \equiv \exists v \in V : (v1, v2) \in E \land \text{has_path}(v, v2)
\]

has a model. The modifiers do not specify that they modify the object where the modifier is called. They just do not exclude this possibility. This is the reason why structural equality \(=\) is used instead of the equality of references to objects \(-\). The latter implies the former. The postcondition of method **wcc** requires more explanation. It returns a set of weakly connected graphs. The union of all these graphs yields the original graph. This is modeled by the predicate \(P\). However, the weakly connected components must be maximal. Therefore, the result is the smallest set \(\hat{x}\) satisfying \(P(\hat{x})\).

The left half of of Fig. 2 contains the implications between the different restrictions of the invariant (true means no restriction of the invariant of directed graphs). Together with Table III this implies that we obtain the specializations in the right half of Fig. 2.

<table>
<thead>
<tr>
<th>Class</th>
<th>Invariant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DG</td>
<td>(E \subseteq V \times V)</td>
<td>general directed graphs</td>
</tr>
<tr>
<td>ADG</td>
<td>(E \subseteq V \times V \land g\text{pred})</td>
<td>acyclic directed graphs</td>
</tr>
<tr>
<td>WC DG</td>
<td>(E \subseteq V \times V \land g\text{pred} \land g\text{acyc})</td>
<td>weakly connected directed graphs</td>
</tr>
<tr>
<td>AWCDG</td>
<td>(E \subseteq V \times V \land g\text{acyc}\land g\text{trans})</td>
<td>acyclic and weakly connected directed graphs</td>
</tr>
<tr>
<td>TREE</td>
<td>(V \subseteq V \times V \land g\text{acyc}\land g\text{trans})</td>
<td>trees</td>
</tr>
</tbody>
</table>

Table II. Classes related to directed graphs, and their invariants.

Figure 2. Implications between structural predicates and specialization hierarchy of directed graphs.

The interesting observation on this example is (cf. Fig. 2) that the implications between the predicates that are used for constructing classes correspond to a specialization hierarchy. In Paragraph NON-POLYMORPHIC SPECIALIZATION HIERARCHIES we will show that this correspondence always holds.

For the following paragraphs we assume a class A and a set of predicates, \(P = \{P_1, \ldots, P_m\}\) on the objects of that class. Each conjunction of these predicates \(P_1 \land \cdots \land P_m\) may be used to define a specialized class. Implications between these conjunctions induce specialization relations, since an object satisfying \(P_1\) automatically satisfies \(P_i\). Each predicate induces a set of objects satisfying the predicate – the implication can be interpreted as a subset relationship. The sets may be disjoint which can be expressed by \(P_i \not\subseteq P_j\). If for two predicates \(P_i\) and \(P_j\)
CONSTRUCTION OF ROBUST CLASS HIERARCHIES

The basic idea is to construct a hierarchy of explicit specializations. In particular, specializations are used non-polymorphically. Let A be a class with predicates $P = \{P_1, \ldots, P_m\}$ on the objects of that class. The idea of this approach (as already indicated by Example 4) is to introduce a class $P_1 \vdash \cdots \vdash P_n, A$ with invariant $I(A) \land P_1 \land \cdots \land P_n$, for every conjunction of predicates $P_1 \land \cdots \land P_n$. Theorem 2 shows that this is always possible.

**Theorem 1** Let A be a class with invariant $I(A)$. Furthermore let $P$ be a closed predicate on the objects of A. Then, there is a class $B$ which is a specialization of $A$ and covariant to $A$ whose objects satisfy $I(A) \land P$.

**Proof:** 7, 9

These specializations may have new methods that can only be called with a meaningful result if a certain property (which leads to a specialization) is true.

**Example 5:** A topological order of a directed graph is only defined if it is acyclic. Therefore, the class $ADG$ contains a method $toporder$ that topologically sorts the directed graph. Suppose this method would be in the class of DG. Then, it can be called only if the precondition $is\_acyclic$ is satisfied. If this is not the case, an exception may be raised or a result may be returned that indicates that the directed graph had cycles. While the former solution leads to a runtime exception which must be caught by the programmer, the latter solution leads to a case distinction – again to be programmed. Both solutions are an additional source of possible runtime errors. Adding the method $toporder$ just to the class $ADG$ of acyclic directed graphs transfers these checks to compile time (in case of a statically type-safe language). Hence, if the programmer accidently calls the method $toporder$ on directed graphs, the compiler recognizes it.

Theorem 1 admits the construction of specialization hierarchies as follows. Consider two subsets $S_1, S_2 \subseteq P$ such that $S_1 \supseteq S_2$. Let $C$ be a class according Theorem 1 that satisfy all predicates of $S_2$, i.e.,

$$\text{Inst}_{C} \equiv \text{Inst}_{A} \land \bigwedge_{P \in S_2} P$$

and $C$ is a specialization of $A$. Then, according to Theorem 1 there is also a class $B$ which is a specialization of $C$ and covariant to $C$ whose objects satisfy additionally the predicates in $S_1 \setminus S_2$, i.e.,

$$\text{Inst}_{B} \equiv \text{Inst}_{C} \land \bigwedge_{P \in S_1 \setminus S_2} P = \text{Inst}_{A} \land \bigwedge_{P \in S_1} P.$$

Since the specialization relation and covariation relation are both transitive, $B$ is also specialization of $A$.

**Example 6:** For directed graphs (cf. Example 4), we have the set of predicates $P = \{is\_acyclic, is\_arc, is\_free\}$. Fig. 3 illustrates the specialization hierarchy. According to the construction principles for the class names at the beginning of this subsection, we should use class names such as $is\_acyclic\_DG$, $is\_arc\_DG$, $is\_free\_DG$ etc. Here, we use the previously defined names (cf. Table II).

The supersets lattice $(P, \supseteq, \emptyset)$ (e.g. those in Fig. 3(b)) induces a specialization hierarchy.

<table>
<thead>
<tr>
<th>$v$</th>
<th>$\text{src}$</th>
<th>$\text{tg}$</th>
<th>$\text{is}_arc$</th>
<th>$\text{is}_acyclic$</th>
<th>$\text{is}_free$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>$v_2$</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>$v_3$</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>$v_4$</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

Table III. Some methods with their preconditions in the classes in Figure 2.
Remark: A finite lattice is a quadruple \((U, \sqsubseteq, \bot, \top)\) where
- \(U\) is a finite set (the universe),
- \(\sqsubseteq\) is a partial order on \(U\) (i.e. \(\sqsubseteq\) is reflexive, anti-symmetric, and transitive),
- \(\bot\) is the smallest element (i.e. \(\bot \sqsubseteq x\) for all \(x \in U\)),
- \(\top\) is the largest element (i.e. \(x \sqsubseteq \top\) for all \(x \in U\)),
- for every two elements \(x, y \in U\) there is a unique least upper bound \(z\) (i.e. \(x \sqsubseteq z, y \sqsubseteq z\) and for all \(w\) with \(x \sqsubseteq w, y \sqsubseteq w\) holds \(z \sqsubseteq w\)), and
- for every two elements \(x, y \in U\) there is a unique greatest lower bound \(z\) (i.e. \(z \sqsubseteq x, z \sqsubseteq y\) and for all \(w\) with \(w \sqsubseteq x, w \sqsubseteq y\) holds \(w \sqsubseteq z\)).

However, this lattice may contain too many classes with the same semantics.

**Example 7:** Example 6 contains the class \(ATDG\) of acyclic directed graphs that are trees. However, every tree is acyclic. Hence, the classes \(ATDG\) and \(TDG\) describe the same objects.

The reason is that in addition to the implications
\[
\bigwedge_{P \in S_1} P = \bigwedge_{P \in S_2} P
\]
for \(S_2 \subseteq S_1 \subseteq P\)
other implications between conjunctions of predicates may hold. This leads to equivalences of some of the conjunctions. It is sufficient to construct only one class in the hierarchy for such an equivalence class. It may even hold \(P_1 \land \cdots \land P_n \Rightarrow \text{false}\). Since it does not make sense to construct classes with an invariant that is equivalent to \(\text{false}\), such classes are excluded. Hence, the number of classes can be reduced.

**Example 8:** In Example 4 we have the following two implications:
\[
\begin{align*}
\text{is}\_\text{tree} &\Rightarrow \text{is}\_\text{acyclic} \\
\text{is}\_\text{tree} &\Rightarrow \text{is}\_\text{wc}
\end{align*}
\]
(i.e. every tree is acyclic and weakly connected). These implications induce several other implications:
\[
\begin{align*}
\text{is}\_\text{tree} &\Rightarrow \text{is}\_\text{acyclic} \land \text{is}\_\text{wc} \\
\text{is}\_\text{tree} &\Rightarrow \text{is}\_\text{free} \land \text{is}\_\text{acyclic} \\
\text{is}\_\text{tree} &\Rightarrow \text{is}\_\text{free} \land \text{is}\_\text{wc}
\end{align*}
\]
CONSTRUCTION OF ROBUST CLASS HIERARCHIES 17

1. Define a set of predicates \( P \) on a class \( A \) and set of implications of \( I \) on conjunctions over the predicates in \( P \).
2. Compute the conjunctive closure of these predicates.
3. Derive the specialization hierarchy from the conjunctive closure.
4. Add some new methods to appropriate specialization classes using the stronger invariant.

The new methods added are methods that become meaningful in specialized classes but not in more general classes, cf. Example 5.

Remark: Often, a specialization can use methods of its generalization without changing its semantics (e.g. all queries such as the method \( V \) returning the vertices of a directed graph).

Hence, these methods can be reused in specializations. Unfortunately, many programming languages (e.g. Java or EIFFEL) do not provide mechanisms for code reuse without subtyping – any subclass is automatically a subtype and can hence be used polymorphically. In languages like Sather or C++ there exist mechanisms for explicit code reuse without subtyping called include- or private inheritance. This reduces the amount of programming and simplifies maintenance of specialization hierarchies considerably.

The disadvantage of non-polymorphic specialization is that it is impossible to access a method that relies on a certain property if the according class does not guarantee that property.

We therefore need conversion methods that convert generalizations to specializations and vice versa:

Let \( SH, P \in P \) be a specialization hierarchy for a class \( A \) over a set \( P \) of predicates, \( B \) be a class of \( SH, P \in P \) be a predicate, and \( B' \in B \) be a class with \( \text{Imp} = \text{Imp}\_B \land P. \) A generalization method \( \text{with} \) predicate \( P \) is a method \( \text{make} \_P : B \rightarrow B' \) that converts an object of class \( B' \) into an object of class \( B \) with the same state, i.e. \( \text{Post}_{\text{make} \_P} B = \text{res} \equiv \text{self} \land \text{res} \neq B \). A specialization method \( \text{with} \) a predicate \( P \) is a method \( \text{make} \_P : B' \rightarrow B \) that converts an object of class \( B' \) into an object of class \( B \) with the same state, provided it satisfies predicate \( P \), i.e. \( \text{Pre}_{\text{make} \_P} B = \text{res} \equiv \text{self} \land \text{res} \neq B \). Thus conversion methods either perform a deep copy of the objects or information must be stored that allow to construct the results of conversion methods. This is analogous to eager and lazy evaluation strategies in functional languages. Specialization methods additionally check the specializing property.

Remark: Formally, the specialization methods for predicates \( P \in P \) ensured by \( \text{Imp} \) are required. They just perform a deep copy without changing the value. Alternatively, the preconditions of these methods could be defined as \( \text{false} \), i.e. they can never be called. We decide not to use this alternative for reasons which become clear in Subsection FACTORIZED CLASSES.

At first glance, it seems that there is another alternative implementation: wrapping the attributes and copying just the wrapper object. However, this is not always possible for specialization methods. Consider the example of graphs, Fig. 4 gives a typical implementation with wrapper objects. The problem is that after the assignment in line (36), it holds \( g1.\text{repr} = g2.\text{repr} \). Hence, adding an edge to \( g1 \) automatically also adds the same edge to \( g2 \) (cf. line (37)). There are only two possibilities to avoid this, either \( g1 \) and \( g2 \) do not share references, or all references to \( g2.\text{repr} \) are destroyed. The latter leaves dangling references. Hence, we choose the former approach.

Example 9: Table V shows the conversion methods for the different classes in the specialization hierarchy shown in Figure 2, their preconditions and return type. The precondition is always \( \text{res} = \text{self} \land \text{res} \neq \text{self} \), i.e. the state is preserved but the object is new. The entry – denotes that for specialization methods just a deep copy is performed and that the generalization method is not present.

The main advantage of the non-polymorphic specialization hierarchy is its robustness: Let

\[ A \text{ be a class, } P \text{ be a predicate on the objects of class } A, \text{ and } m \text{ be a method that can be called only if } P \text{ is satisfied. Suppose } m \text{ is called accidentally on an object } o \text{ where } P \text{ is not satisfied.} \]

\[ \text{Then, in the non-polymorphic specialization hierarchy, } m \text{ is not a method of the class of } o. \]

Compilers for typed object-oriented languages (e.g. C++, EIFFEL, JAVA, SATHER) refuse such programs. However, there is a price to pay. In the above scenario, the object \( o \) may satisfy \( P \) although its class does not guarantee this. Then, method \( m \) cannot be called on \( o \) either.

The object \( o \) must be converted before into an object to a specialized class that guarantees the satisfaction of \( P \). Thus, the solution is not flexible. Assuming that a deep copy costs time \( O(n) \) where \( n \) is the size of the object to be copied, at least the asymptotic complexity of a program is not increased. However, the constant factors may increase considerably.

Immutables

The goal of a specialization hierarchy of immutable classes is to convert the non-polymorphic specialization hierarchy into a polymorphic specialization hierarchy. The basic idea is that each method is a function returning a value and does not change the state of \( o \), i.e. every class in the hierarchy is immutable. If the state of an object cannot be changed, invariant properties are not affected. This leads to the following result:

Theorem 3 Let \( A \) be an immutable class with a predicate \( P \). Then, there is an immutable class \( B \) conforming to \( A \) whose objects satisfy \( \text{Imp} = \text{Imp} \_A \land P \).
CONSTRUCTION OF ROBUST CLASS HIERARCHIES

Table V. Conversion methods for directed graphs.

Table VI. Modifiers in the hierarchy of immutable classes of directed graphs.

**Example 11:** The class $\overline{\text{DG}}$ should have a method $\text{connect}(v_1, v_2 : \overline{\text{VERTEX}})$ with precondition $\text{Pre}_{\text{connect}}(v_1, v_2 : \overline{\text{VERTEX}}) \land \neg \text{hasPathPathPath}(v_2, v_1)$ that preserves $\overline{\text{DG}}$.

These methods can be defined inductively, starting with the highest class in the conformance hierarchy down to the lowest. Using the following rules for every method $m$ with a return type $C$ of the conformance hierarchy (except specialization methods), we can construct the versions of $m$ in a class $B$ of the conformance hierarchy, where $\overline{\text{DG}} \models \text{Pre}_{\overline{B}}$. For $i = 1 \cdots n$, $P_i$ are:

- (i) $B$ contains all versions of $m$ contained in one of its subclasses.
- (ii) If for all superclassess $D$ of $B$, there is no version $m'^3$ of $m$ that preserves $\overline{\text{DG}}$, then $B$ contains a variant $m_1 \cdots m_n$ with the same parameters as $m$ and return type $\overline{D}$, $\text{Pre}_{\overline{D}} \models \text{Pre}_{\overline{B}} \land \neg \text{hasPathPathPath}(v_2, v_1)$.

The proof is analogous to 3.5.

**Example 12:** Table VII shows variants of methods by simply denoting the return type for the classes that provide the method. If an entry on row $m$ in column $X$ is empty, then $m$ is not a method of class $X$. The knowledge which properties are preserved can be coded in the return type $E$. E.g. the immutable class $\overline{\text{DG}}$ contains the method $\text{addVertex}(v : \overline{\text{VERTEX}}) : \overline{\text{DG}}(\overline{\text{VERTEX}}, \overline{\text{EDGE}})$ because the acyclicity of a direct graph is preserved when the modifiers. The methods always return an object of class $\overline{\text{DG}}$. Their preconditions are the same as in the class $\overline{\text{DG}}$ of directed graphs. This is true for all classes in the immutable hierarchy, otherwise the conformance would be violated. The pre- and postconditions for the queries are the same as in the class $\overline{\text{DG}}$ of directed graphs, because they do not change the state.

The counterparts of the methods $\text{connect}(v_1, v_2 : \text{VERTEX})$ and $\text{addVertex}(v : \text{VERTEX})$ of the classes in the non-polymorphic specialization hierarchy, are the methods $\text{connect}(v_1, v_2 : \overline{\text{VERTEX}})$ and $\text{addVertex}(v : \overline{\text{VERTEX}})$ of the conformance hierarchy.

**Example 10:** Using the same predicates $\overline{\text{DG}}$, we consider again the example of directed graphs. To distinguish the classes considered here from the classes in the non-polymorphic specialization hierarchy, class names will be overlined by convention. Using the same predicates $\overline{\text{DG}}$, a conformance hierarchy of immutable classes analogous to the specialization hierarchy shown in Figure 2 can be established. The difference is that all the classes are immutable and the relation is the conformance instead of the specialization. Table VI shows the postconditions for the modifiers.

Table VII. Table VII shows variants of methods by simply denoting the return type for the classes that provide the method. If an entry on row $m$ in column $X$ is empty, then $m$ is not a method of class $X$. The knowledge which properties are preserved can be coded in the return type $E$. E.g. the immutable class $\overline{\text{DG}}$ contains the method $\text{addVertex}(v : \overline{\text{VERTEX}}) : \overline{\text{DG}}(\overline{\text{VERTEX}}, \overline{\text{EDGE}})$ because the acyclicity of a direct graph is preserved when

29/10/1999 11:16 PAGE PROOFS text
inserting a vertex. There are no variants of method sym_plus because it is impossible to preserve the acyclicity when the graph has an edge. Similarly, the method trans_plus preserves all properties except of being a tree.

The preconditions of the variants are the same as in the non-polyomorphic specialization hierarchy because it desired to preserve predicates such as acyclicity, weak connectivity etc. The postconditions are the same as their counterparts (cf. Table VI).

Thus, the hierarchy of immutable classes for a class A over predicates P can be designed by the following algorithm:

1. compute the conjunctive closure of P
2. make A immutable
3. assign to each element in the conjunctive closure a unique class
4. traverse the conjunctive closure from the highest to the lowest class
   (a) form subtype relationships by conforming inheritance between direct neighbors in the hierarchy
   (b) construct the variants of methods where applicable
   (c) add methods that rely on the stronger invariant of this class

Alternatively, a non-polyomorphic specialization hierarchy can be transformed into a hierarchy of immutable classes by making all methods immutable. In case of specialized modifiers, the corresponding variant of the method must be chosen.

The immutable hierarchies are robust and flexible. The main problem with the hierarchy of immutable classes is the efficiency of its implementations: constructing an object of size m may cost time O(m^2) even if it could be constructed in linear time, because every constructor copies an object.

Factorized Classes

The main idea of this design principle is to convert a non-polyorphic specialization hierarchy into a polyomorphic hierarchy by a projection to those methods that do not violate the conformance condition in the specializations of that class. E.g., the pre- and postconditions of methods for computing the set of vertices and edges, path problems etc. will be the same for all directed graphs. For this reason, we define the factorization of two classes A and B as a class C with the following properties: (i) \( \text{Inv}_C = \text{Inv}_A \lor \text{Inv}_B \), (ii) C contains a method \( m \) iff \( m \) is contained in \( A \) and in \( B \), (iii) \( \text{Pre}_{m,C} = \text{Pre}_{m,A} \lor \text{Pre}_{m,B} \), and (iv) \( \text{Post}_{m,C} = \text{Post}_{m,A} \lor \text{Post}_{m,B} \). It is not hard to show that A conforms to C and B conforms to C. For pragmatic reasons, we exclude \( m \) from C iff \( \text{Pre}_{m,C} \implies \exists \text{ Post}_{m,C} \implies \text{true} \). The generalization to the factorizations of \( n \) classes \( A_1, \ldots, A_n \) is obvious. In \( \forall \) we have shown the following theorem:

**Theorem 5** Any specialization hierarchy induces a conformance hierarchy of classes, where for each class A in the specialization hierarchy, there is a class \( A' \) in the conformance hierarchy which is a factorization of all specializations of A.

**Example 13:** Table VIII shows the factorized classes of the specialization hierarchy shown in Figure 2. Trees are omitted in the table, since the class of trees is factorized from only one class. Figure 5 shows the conformance hierarchy of these classes.

The factorized classes also contain the conversion methods:

**Lemma 1** Let SH be a specialization hierarchy of class A over predicates P and SH' the factorized hierarchy. Then, for all classes B of SH the following two properties hold for the factorized class B':

(i) For each specialization method make\_P : C there is method make\_P : SC' in B' and (vice versa)

(ii) For each generalization method forget\_P : C there is method forget\_P : SC' in B' and (vice versa),

where C is a class of SH and C' is its factorized class.
Proof: Let $P \in \mathcal{P}$ be an arbitrary predicate. For all classes of $SH$, the precondition of $\text{make } P$ is $P$. Thus $\text{make } P$ occurs in all of the factorized classes. The postcondition is always the same. It is easy to see that the return type can be polymorphic. A similar argument applies for the generalization methods.

Remark: Lemma 1 is a reason why every specialization method must be present even if the property is already ensured. Otherwise, these methods would not occur in the factorized hierarchy, which implies that conversions cannot be done when the factorized classes are used polymorphically.

The hierarchy of factorized classes is robust. It is also more flexible than the non-polymorphic specialization hierarchy because it is polymorphic. However, it is less flexible than the hierarchy of immutable classes because the factorized classes contain fewer methods. Factorized classes have the same efficiency properties as the non-polymorphic specialization hierarchy. However, there is a problem when extending a hierarchy of factorized classes.

Hence, extensions of hierarchies of factorized classes may remove methods from classes. Therefore, application programs using hierarchies of factorized classes may be invalidated by such extensions.

Dynamic Properties

Let $A$ be a class with predicates $P \equiv \{P_1, \ldots, P_m\}$ on the objects of the class. A common solution for construction of specializations is to maintain dynamically the specialized properties using dynamic predicates. The whole specialization hierarchy collapses into one class $\mathcal{A}$. A method requires that some predicates of $P$ are satisfied. $m$ must raise an exception if this precondition is not satisfied in order to avoid runtime errors that point deeply into the internal calling hierarchy of a library.

Since every predicate $P \in \mathcal{P}$ is zero-ary, the result of the last check can be remembered by storing it in a Boolean attribute $P'$. The basic idea is that $A$ ensures the invariant $P = \text{true } \iff \text{true } \Rightarrow P$. This can be achieved by setting the attribute to true whenever predicate $P$ is called and returns true, i.e. $\text{Post}_{P',A} \equiv \text{Post}_{P',A} \land P' = \text{res}$ and for each method $m$ of $A$ which does not preserve predicates $P_1, \ldots, P_m \in \mathcal{P}$ by setting the attribute $P'$ to false, i.e.

![Figure 5. Conformance hierarchy of the factorized classes.](image)

### Table IX. Postconditions of Modifiers for Directed Graphs with Dynamic Properties

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{make } V, E$</td>
<td>$V \equiv V' \land E \equiv E' \land \forall v \in V: \neg \exists x \in E: v \xrightarrow{x} u \land \forall v \in V: \neg \exists x \in E: v \xrightarrow{x} u$</td>
</tr>
<tr>
<td>$\text{add}_2$</td>
<td>$V \equiv \text{add}_2(V', E') \land \forall v \in V: \neg \exists x \in E: v \xrightarrow{x} u \land \forall v \in V: \neg \exists x \in E: v \xrightarrow{x} u$</td>
</tr>
<tr>
<td>$\text{connect}(v_1, v_2)$</td>
<td>$V \equiv \text{connect}(v_1, v_2) \land \forall v \in V: \neg \exists x \in E: v \xrightarrow{x} u \land \forall v \in V: \neg \exists x \in E: v \xrightarrow{x} u$</td>
</tr>
<tr>
<td>$\text{del}_2$</td>
<td>$V \equiv \text{del}_2(V', E') \land \forall v \in V: \neg \exists x \in E: v \xrightarrow{x} u \land \forall v \in V: \neg \exists x \in E: v \xrightarrow{x} u$</td>
</tr>
</tbody>
</table>

For all classes of $DG$, the postcondition of $\text{make } A$ is $A = \text{false } \land P = \text{false } \land P = \text{false } \land P = \text{false } \land P = \text{false }$. These definitions ensure that $A$ conforms to $A$. This design does not lead to a robust implementation since all property checks are performed at runtime and method availability is also decided then.

Example 15: The invariant of the class $\mathcal{DG}$ of the class $DG$ with dynamic maintenance of properties is:

$\text{Post}_{\text{DG}} = \text{Post}_{\text{DG}}' \land P = \text{false } \land \text{Post}_{\text{DG}}' = \text{false } \land P = \text{false } \land P = \text{false } \land P = \text{false }$. These definitions ensure that $A$ conforms to $A$. This design does not lead to a robust implementation since all property checks are performed at runtime and method availability is also decided then.

Since every predicate $P \in \mathcal{P}$ is zero-ary, the result of the last check can be remembered by storing it in a Boolean attribute $P'$. The basic idea is that $A$ ensures the invariant $P = \text{true } \iff \text{true } \Rightarrow P$. This can be achieved by setting the attribute to true whenever predicate $P$ is called and returns true, i.e. $\text{Post}_{P',A} \equiv \text{Post}_{P',A} \land P' = \text{res}$ and for each method $m$ of $A$ which does not preserve predicates $P_1, \ldots, P_m \in \mathcal{P}$ by setting the attribute $P'$ to false, i.e.

Discussion

Table X summarizes the non-functional properties of the different solutions to deal with specialization hierarchies. The dynamic properties solution is the most flexible solution, the non-polymorphic specialization hierarchy the least flexible since no polymorphism is allowed. The immutable classes require copying when modifying (constructing) objects, while this is not required for the dynamic predicates, e.g. if $g \text{connect}(v_1, v_2)$ is called in the class $DG$
of immutable classes, \( g \) must be copied and the edge is added to the copy of \( g \). The non-polymorphic hierarchy and the factorized classes only require copying when specializing or generalizing an object. Thus, these solutions are never less efficient than the immutable classes. Since dynamic properties require the evaluation of predicates at runtime, the efficiency of this solution is incomparable with the non-polymorphic specialization hierarchies and factorized hierarchies. Dynamic properties are not robust since the other solutions demonstrate that the properties can be ensured statically, i.e., the dynamic properties may lead to runtime errors which could be avoided. The factorized classes are not safely extensible: adding a new property extends the specialization hierarchy and may lead to fewer methods in the factorized classes. Thus, a program using an extended library may be illegal although a program using the original library is legal.

Dynamic predicates are the most economic solution from an implementation point of view: the overall size of the other solutions (in terms of length) can be exponential in the size of the dynamic predicates. This is the reason why the solution of dynamic predicate is often chosen. We sketch in Section GENERATION OF CLASS HIERARCHIES how all solutions can be generated from specifications that are not significantly longer than the dynamic predicates. This makes non-polymorphic specialization hierarchies, factorized hierarchies, and hierarchies of immutable classes as feasible as the dynamic properties.

**Remark:** In practice the design alternatives described above are often mixed, i.e. conformance hierarchies are combined with dynamic properties.

### Conformance Hierarchies of Generic Classes

This subsection provides an approach to construct conformance hierarchies of generic classes. The approach is similar to the construction of non-polymorphic specialization hierarchies. Instead of predicates of classes, properties of instances of the parameters of generic classes (type bounds) are considered. The basis for the construction method is the following theorem:

**Theorem 6** Let \( A(T_1, \ldots, T_k) \) be a generic class and \( P_1(T_1, \ldots, T_k) \) a predicate, that reduces the set of wild generic parameters. Let \( B(T_1, \ldots, T_k) \) be a generic class, where its instantiations are also instantiations of class \( A(T_1, \ldots, T_k) \), and hold the stronger predicate \( P_2(T_1, \ldots, T_k) \Rightarrow P_1(T_1, \ldots, T_k) \). Then \( B \) conforms to \( A \).

**Proof:** 7.5.8

The predicates in Theorem 6 are type bounds. Type bounds can be viewed as predicates on the parameters of generic classes. Thus, we obtain a similar approach as in Paragraph NON-POLYMORPHIC SPECIALIZATION HIERARCHIES OF SECTION DESIGN OF SPECIALIZATION HIERARCHIES using the predicates on type bounds instead of the predicates on objects.

**Example 16:** Suppose the class \( \text{SET} \) has the following modifiers:

- \( \text{init} \) creates an empty set
- \( \text{insert}(x) \) inserts element \( x \) into a set

\( \frac{29/10/1999 11:16 PAGE PROOFS text}{29/10/1999 11:16 PAGE PROOFS text} \)
Theorem 7 Let $A(T_1, \ldots, T_k)$ be a generic class and $P_1, \ldots, P_m$ predicates over $T_1, \ldots, T_k$. Then the lattice $(\Pi(P_1, \ldots, P_m), \Rightarrow, T_1 \land \cdots \land T_m, \text{true})$ induces a conformance hierarchy $\langle A, \text{conforms to, } \subseteq, T \rangle$ with $A = \{ Q(A(T_1, \ldots, T_k) : Q \in \Pi(P_1, \ldots, P_m) \}, \subseteq = P_1 \land \cdots \land P_m A(T_1, \ldots, T_k) \}$ and $\subseteq = A(T_1, \ldots, T_k)$.

Our design approach for hierarchies of conforming subclasses is therefore analogous to the construction of the non-polymorphic specialization hierarchy:

1. define a set of predicates on the parameters $T_1, \ldots, T_k$ of a generic class $A(T_1, \ldots, T_k)$.
2. compute the conjunctive closure of these predicates.
3. derive the conformance hierarchy from the conjunctive closure.
4. add some new methods to the subclasses, which use the stronger assumptions on the instance of generic parameters.

The conformance hierarchies constructed by the above approach are flexible, efficient, safely extensible, and robust. The reason for the flexibility is that conformant subclasses can be used polymorphically. There is no requirement such as immutability on the classes. Thus, there is no reason that prevents efficient implementations of classes in the conformance hierarchies. The conformance hierarchy is for the same reason safely extensible as the non-polymorphic specialization hierarchies and the immutable classes. Theorems 6 and 7 ensure the robustness of the conformance hierarchy.

Table XII. Post Conditions of Methods in Set Classes

<table>
<thead>
<tr>
<th>Method</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>$y = \text{null} \Rightarrow y = x$</td>
</tr>
<tr>
<td>delete</td>
<td>$y = \text{null} \Rightarrow y = x$</td>
</tr>
<tr>
<td>add</td>
<td>$y = \text{null} \Rightarrow y = x$</td>
</tr>
</tbody>
</table>

Figure 6. Properties of universes and corresponding hierarchy of conformant subclasses.

Implementations

In Subsections Design of Specialization Hierarchies and Conformance Hierarchies of Generic Classes, the structure of hierarchies of abstract classes has been discussed. This subsection changes the focus to implementation classes. In particular, we demonstrate how knowledge of the specialization and conformance hierarchies may allow for more efficient implementations while retaining robustness.

Let $A$ be an abstract (generic) class. An implementation $A'$ of $A$ consists of two parts: the implementation of the data structure specified by $A$ and the implementation of the methods. The data structure is implemented by some attributes and is called representation. Suppose, the class $A$ is specified by its invariant $\text{Inv}_A$ and for each method $\text{method}$ by its precondition $\text{Pre}_{\text{method}}$ and its postcondition $\text{Post}_{\text{method}}$. Then $A'$ must also satisfy the invariant $\text{Inv}_{A'}$ and the postcondition $\text{Pre}_{\text{method}} \land \text{Post}_{\text{method}}$ for each method $\text{method}$ of $A$. This is certainly satisfied if $A'$ conforms to $A$. Therefore, it is reasonable to decide that every implementation of an abstract class $A$ is a conforming subclass of $A$, i.e. $A'$ inherits $A$ with the conformance inheritance relation. This has the additional advantage that for an object $x : A$, any implementation for $x$ can be chosen for $x$, but the actual decision is hidden.

Example 17: Class $DG_{ADJLIST}$ is an implementation of class $DG$ using an adjacency lists, cf. Fig. 7. This class represents directed graphs by an array of its vertices (vertices), the reverse mapping from vertices to its array element indices (map), and the adjacency lists which are represented by an array of lists (edge). The class $DG_{ADJLIST}$ uses the generic class $MAP$ to define finite mappings from objects to objects $T$. It has methods for checking whether an object of class $T$ is in the domain of the mapping (defined), for adding an association (add), and for getting the object of class $U$ associated to an object of class $T$ (lookup). The class $ARRAY$ is predefined and represents flexible arrays, i.e. they can be extended dynamically. The constructor must specify its initial size. The class $LIST$ contains methods for inserting elements into a list (insert), for navigating within the list (next), for setting a next position (next). The class $DG$ inherits $DG_{ADJLIST}$ by generalizing the class $DG_{ADJLIST}$ to a generic class, i.e. $DG$ inherits $DG_{ADJLIST}$ with $DG_{ADJLIST}$ as the concrete class and $DG$ as the abstract class.

The postconditions of the methods make, $V$, and $E$ are:

- $P_{\text{Post}_{\text{make}}_{DG_{ADJLIST}}}$: $V \equiv V' \land E \equiv E' \land V.Vs_{\text{empty}} \land E.E_{\text{empty}}$
- $P_{\text{Post}_{\text{add}}_{DG_{ADJLIST}}}$: $\text{unchanged} \land \forall v : \text{VERTEX} : \text{map} \text{defined}(v) \Rightarrow \text{vertices}[\text{map.lookup}(v)] = v$
- $P_{\text{Post}_{\text{get}}_{DG_{ADJLIST}}}$: $\text{unchanged} \land \forall v : \text{VERTEX} : \text{map} \text{defined}(v) \Rightarrow \text{edges}[\text{map.lookup}(v)] = \{ e \in \text{edge} : \text{vertex} \in \text{map.lookup}(v) \}$

The postconditions of make specifies that as a side-effect the arguments become empty. This is possible because the postcondition of class $DG$ does not specify that the arguments remain unchanged. The postcondition of $V$ states that the result contains all vertices that are defined by $\text{map}$. Together with the invariant this implies that it returns all vertices stored in the array $\text{vertices}$. The postcondition of $E$ specifies that an edge in the result is if it is somewhere in an adjacency list. For the remaining methods we assume the same pre- and postconditions as in the class $DG$.  

29/10/1999 11:16 PAGE PROOFS text
CONSTRUCTION OF ROBUST CLASS HIERARCHIES

The class DG_ADJLISTS conforms to class DG. The implications on the pre- and postconditions are straightforward to prove. The implications on the invariant use the postcondition of V and E respectively. It remains to prove Inv_DG_ADJLISTS \Rightarrow Inv_DG. Let e : EDGE be an edge. It is sufficient to show e \in E \Rightarrow e \in V \land e \in E \land e \in V. This implication can be proven as follows:

e \in E \Rightarrow map_defined(e.src) \land e \subseteq [\text{map.lookup(e.src)}] \land member(e) \land \text{map_defined(e.dest)} \land e \subseteq [\text{map.lookup(e.dest)}] \land member(e) \land \text{map_defined(e.dest)}

An implementation class DG_ADJMATRIX using a Boolean adjacency matrix instead of adjacency lists can be constructed analogously.

Methods of an abstract class A can be classified as follows: kernel methods are methods without an implementation and derived methods are methods with an implementation. Derived methods must not rely on a particular implementation of an abstract class and can therefore be implemented in the abstract class.

Example 18: Suppose the methods init, make, V, and E of directed graphs are abstract. Then all other methods of directed graphs are derived. Fig. 8 shows some of the modifiers and queries. The implementations are valid for every representation of directed graphs. The kernel methods init, make, V, and E can only be implemented using a concrete representation for graphs. If all methods of class DG except init, make, V, and E have an implementation in class DG, then the class DG_ADJLISTS in Fig. 7 is already a complete implementation. However, it is reasonable to override methods such as add_vertex, connect, del_edge and del_vertex.

The advantage of using inheritance hierarchies of abstract classes is that implementations can use stronger invariants or stronger requirement on instances of the parameters of a generic class. This affects representations as well as algorithms implementing methods. In particular, specializations might allow the use of more efficient algorithms to implement some methods.

Example 19: Consider the abstract class ORDERED_SET(T < ORDERED(T)). Since the universe is ordered, it is possible to implement this class by binary search trees. This is not a legal implementation for SET(T) since there are instances of T without a total order on T (which is required for binary search trees). Consider the abstract class PLANAR_GRAPH(V < VERTEX, E < EDGE) determining the crossing number of an arbitrary graph in is graph is in general NP-complete, while it can be solved in time O(|V| + |E|) on planar graphs. Every implementation of planar graphs can make use of this knowledge, but in general, this efficient implementation is impossible.

Implementation Hierarchies

The approach of Subsection IMPLEMENTATION is rather inflexible. If the representation of an abstract class is fixed it cannot be changed without inheriting the implementation class. The same holds for algorithms implementing methods. This subsection sketches how to implement classes such that representations as well as algorithms can be exchanged even during runtime of programs without robustness violations. This achieved by combining the two design patterns bridge and strategy. An object-oriented design pattern consists of several collaborating classes which represent a reusable solution scheme to a design problem. In our case it is the dynamic exchange of representations of abstract classes and implementations of derived methods. For

Figure 7. Representation of Directed Graphs by Adjacency Lists

Figure 8. Some Derived Methods for Directed Graphs
A strategy defines a family of algorithms solving the same problem and makes them interchangeable. This can be achieved by defining an abstract class $P$ for the problem solved by the different algorithms and by defining for each algorithm $A$ its own class $A$ which must be a conformant subclass of $P$. If one has now a polymorphic algorithm object $x$ of type $P$, then an algorithm can be exchanged by creating an object of the desired algorithm class and assigning it to $x$. The design pattern bridge for an abstract class $A$ allows the dynamic exchange of the implementations of $A$.

**Example 20:** A bridge for the class $DG($VERTEX, EDGE$)$ allows the replacement of a set implementation of an object $x$: $SDG($VERTEX, EDGE$)$ by an implementation based on adjacency lists or adjacency matrices. This representation change exchanges a class—including all its methods. It does not yet allow the dynamic exchange of algorithms. Therefore we combine the design pattern Bridge with Strategy into the combined pattern dynamic bridge which allows dynamic exchange of representations as well as algorithms.

**Example 21:** Fig. 9 shows the dynamic bridge for directed graphs. The attribute repr stores the current representation for direct graphs. It can be changed using the method change_repr, cf. Fig. 9. The initialization method initializes the representation and the algorithms to default implementations. The basic idea is that the most general algorithm classes (cf. lines (32)–(43)) delegate the algorithm execution to the current representation (cf. also lines (25) and (29)). The kernel methods are also delegated to the current representation. Therefore, after creation of an object of class $DG($VERTEX, EDGE$)$, it executes the same methods as in the default implementation class (here the class $DG($SET, SET$)$ that represents directed graphs explicitly by sets).

Fig. 10 illustrates a scenario of using the dynamic bridge in Fig. 9. In line (3) a new dynamic bridge object is created. The procedure calls in line (5) build a graph consisting of vertices 1 and 2 and edge (1, 2). Consider for example the call $dg.connect(1, 2)$, it executes line (25) in Fig. 9, i.e. $connec_t_dg.connect(1, 2, repr)$. This call executes line (35) in Fig. 9, i.e. $repr.connect(1, 2, repr)$, because the object referenced by $connect_dg$ is an object of class $CONNECT($INT, $INT$), $SDG($INT, $INT$), $MYDG($INT, $INT$). Since the implementation class $DG($SET, SET$)$ is the current class of $repr$, this call executes the method $connect$ implemented in $DG($SET, SET$)$, $MYDG($INT, $INT$). Analogous arguments show that the call in line (6) of Fig. 10 executes the method $wcc$ implemented in class $DG($SET, SET$)$, $MYDG($INT, $INT$). After line (7), $MYWCC$ is the class of the object referenced by $wcc_dg$. This assignment is legal, because $MYWCC$ is a subtype of $WCC$ (cf. lines (14)–(18), Fig. 10). The call in line (8) executes line (29), i.e. $wcc_dg.wcc(repr)$. Since $MYWCC$ is the current class of $wcc_dg$, the method $wcc$ of class $MYWCC$ is called. Line (9) of Fig. 10 changes the representation to adjacency matrices without changing its abstract state, i.e. $dg$ still represents a directed graph with vertices 1 and 2 and edge (1, 2). After line (9), $DG($ADJ, MATRIX$)$ is the dynamic type of the object referenced by $repr$. Hence, with the same arguments as above, the call in line (10) executes the method $connect$ of class $DG($ADJ, MATRIX$)$. However, the call in line (11) still executes method $wcc$ of class $MYWCC$ because this is still the class of the object referenced by $wcc_dg$.

**Further Design Patterns:**

- an initialization,
- a procedure for changing the representation,
- kernel methods,
- derived methods, and
- general algorithm classes.

It contains an attribute $repr : \langle A \rangle$ containing the current representation and for each derived method $m$ an attribute $m_{alg} : \langle m \rangle$ storing the current algorithm implementing $m$. The initialization initializes each of these attributes to a default implementation. The attributes for derived methods are initialized with objects of the corresponding general algorithm classes (cf. lines (9)–(14) in Fig. 9). A representation change performs an abstract deep copy of the representation object (cf. lines (3)–(5) of Fig. 9). Kernel methods simply delegate the execution to the current representation. If the method is a procedure, it is executed on $repr$ (cf. lines (15)–(17) in Fig. 9), if it is a function the result on the execution on $repr$ is returned (cf. lines (18)–(23) in Fig. 9). Derived methods delegate the execution to the corresponding algorithm object, e.g. a method $m_{alg}$ delegates the execution to the algorithm object $m_{alg}$. It passes the current representation $repr$ as an argument. If the method $m_{alg}$ is a procedure, it is executed on the algorithm object (cf. lines (24)–(26) of Fig. 9), if it is a function it returns the result of its execution on the algorithm object (cf. lines (28)–(30) of Fig. 9). A general algorithm class $M$ for method $m$ has the same generic parameters as class $A$ and additionally a parameter $REP : A$. It contains the method $m$ which has the same parameters as the method $m_{alg}$ in $A$ and additionally a parameter $repr : REP$. Its body simply delegates the execution to the object $repr$ (cf. lines (32)–(43) of Fig. 9). A specific algorithm for the implementation of $m$ is implemented in a class $MYM$ that inherits from the general algorithm class $M$. Assigning a $MYM$-object to the algorithm object $m_{alg}$ redirects the execution to the specific algorithm.

In general, it is not possible for robustness reasons to access all possible implementations for $m_{alg}$ by means of a dynamic bridge.
Example 22: There is an algorithm for computing the transitive closure of a graph that is based on successively squaring its adjacency matrix. Because of its dependency on an adjacency matrix representation, this algorithm cannot be used if the current representation is different, e.g. the set implementation presented here.

We therefore distinguish representation-dependent from representation-independent algorithms. In general, representation-dependent algorithms must not be used if the current representation does not fit. As the library designer cannot predict anything about the current representation of an abstract class in a dynamic bridge (it is possible to change the representation at any time), we only allow the use representation-independent algorithms. E.g., in Fig. 9 this is achieved by the fact that only representation-independent algorithms can be defined by algorithm classes which are subtypes of the algorithm attribute types (e.g. \( WCC(\text{VERTEX}, \text{EDGE}) \)).

However, efficient algorithms implementing a method are often representation-dependent. In order to use these algorithms, the concrete representation must be known statically. We therefore define a static bridge where — in contrast to the dynamic bridge — the representation now becomes a generic parameter of the bridge. For static bridges, the representation is fixed at object creation time. However, it is then possible to use representation-dependent algorithms for the implementation of derived methods.

Example 23: Fig. 11 shows the static bridge for directed graphs. The main difference is that static bridges have an additional generic parameter \( \text{REPR} \) (cf. line (1)). The instances for this generic parameter must conform to the class \( DG(\text{VERTEX}, \text{EDGE}) \). The representation object \( \text{repr} \) of type \( \text{REPR} \) (line (2)). In contrast to the dynamic bridge in Fig. 9, there is no method for changing the representation. The most general algorithm classes are instantiated with the generic parameter \( \text{REPR} \) (cf. lines (3)-(7)), i.e. they use exactly the same representation as the object \( \text{repr} \). The initialization method changes accordingly (cf. lines (8)-(15)). A representation-dependent algorithm may inherit from the most general algorithm class with an adequate instantiation (cf. lines (18)-(23)). All methods except the initialization are implemented as in the dynamic bridge.

Remark: A static bridge for abstract class \( A \) does not require that implementations of an abstract class \( A \) are subtypes of \( A \). Therefore, it is possible to use directly implementations of \( A \) from other libraries. Dynamic bridges and the approach in Subsection IMPLEMENTATION require that every implementation is a subtype of \( A \). Therefore, either every implementation \( A' \) of \( A \) from other libraries have to be changed by making \( A' \) a subtype of \( A \) or a new class is defined which is a subtype of \( A \) and \( A' \).

Figure 10: Changing Dynamically Representations and Algorithms of Directed Graphs

In summary, dynamic bridges for a class \( A \) allow the dynamic exchange of implementations of \( A \) as well as algorithms implementing particular methods. These algorithms, however have to be representation-independent. In contrast, a static bridge for \( A \) allows representation-dependent algorithms at the price of not being able to exchange the implementation of \( A \) at runtime.

Summary

We demonstrated that specialization hierarchies can be constructed systematically. There are four different solutions. Each solution lacks a non-functional property: the non-polymorphic specialization hierarchy is not flexible, the factorized hierarchy is not safely extensible, the hierarchy of immutable classes may prevent efficient implementations, and the solution with dynamic properties is not robust. It seems that there is no solution to a specialization hierarchy which satisfies all those four properties. Hence, a class library of algorithms and data structures must offer all those solutions, because a library designer cannot decide which properties users prefer.

The introduction of bridges allows interchangeability of implementations of data structures and algorithms implementing derived methods. Again, there is a tradeoff between flexibility and efficiency. Static bridges require a fixed implementation of a data structure and allow interchangeability of representation-dependent algorithms. In contrast, dynamic bridges allow interchangeability of data structure implementations, but only interchangeability of the possibly less efficient representation-independent algorithms. In some cases independent algorithms need not even exist, which means that they can only be exchanged within static, but not within dynamic bridges. Hence, a library of algorithms and data structures must provide both bridges.

We conclude that all concepts discussed in this section should be provided by robust libraries of algorithms and data structures. Fig. 12 illustrates the complete architecture of our directed graph example. It indicates that the construction and maintenance of a robust class library becomes quite expensive if all the concepts described in this subsection are provided, since the
number of classes grows exponentially. Due to the large number of classes, it also becomes hard to use such a library. We propose to solve the latter problem by an interactive user interface. Because the structure arises in a systematic way, the user interface can provide compact information on specialization and conformance hierarchies. In particular, it must offer

- choices of the four non-functional properties for choosing the desired solution,
- choices for the predicates used for construction of the class hierarchy, and
- choices for methods of the classes in the class hierarchy.

If users specify the particular information, it is possible to select all classes in a specialization/conformance hierarchy satisfying the specified properties. Bridges can be instantiated either by default, or may be supported by a graphical interface offering a choice between all possible algorithms (and representations in the case of dynamic bridges).

**Example 24:** Consider the class $DG$ of directed graphs and its method $trans_{clos}$ computing the transitive closure. Let $DG_{ADJMATRIX}$ be its implementation by adjacency matrices and $DG_{ADJLIST}$ its implementation by adjacency lists. Suppose there is an algorithm $SQUARE_{ADJMATRIX}$ for computing the transitive closure by squaring adjacency matrices, an algorithm $TRANS_{CLOS,LIST}$ for computing the transitive closure based on adjacency list, and an algorithm $TRANS_{CLOS,GEN}$ computing the transitive closure independently of the representation. If there is not yet made a decision for an implementation of directed graphs, the choice of algorithm $SQUARE_{ADJMATRIX}$ fixes the implementation to adjacency matrices (see Figure 13). If the user fixes the implementation to adjacency matrices, then the user interface must offer only algorithms $SQUARE_{ADJMATRIX}$ and $TRANS_{CLOS,GEN}$ for method $trans_{clos}$ (see Figure 14).

Next, we focus on supporting the task of constructing and maintaining robust libraries efficiently. The goal to be attacked in the next section is to generate class hierarchies from compact specifications, therefore allowing for interactive tools aiding the generation process.

**GENERATION OF CLASS HIERARCHIES**

The large number of required classes in robust class hierarchies causes several problems:

- understanding the library structure and usage become hard,
- construction is expensive and time consuming, and
- maintenance becomes almost impossible.

This section discusses our solution to these problems, which is based on the construction principles described in **INHERITANCE HIERARCHIES**. In Subsection **SPECIALIZATION HIERARCHIES**, we sketch how to generate specialization hierarchies, followed by the generation of conformance hierarchies in **CONFORMANCE HIERARCHIES**. Finally, in **SUBSECTION BRIDGES AND ALGORITHM CLASSES** we sketch how to generate bridge classes from abstract classes.

**Specialization Hierarchies**

This section provides an approach how to generate specialization hierarchies from short specifications. The goal is that these specifications should be much shorter than the source code of the generated classes. The generation algorithm itself is largely based on automating the design method derived from Theorem 2. The construction method for non-polymorphic specialization hierarchies and the factorized hierarchies is based on previous work of the authors. Here, we also sketch how to generate hierarchies of immutable classes and classes with dynamic properties. In particular, we slightly extend the specification language described in and demonstrate that all specialization hierarchies described in Subsection **DESIGN OF SPECIALIZATION HIERARCHIES** can be generated from these specifications.

Before we demonstrate generation techniques, we have to define the requirements for an adequate specification. The specification for a specialization hierarchy must contain enough information to derive at least the following information:

1. The signature, implementations and postconditions of all methods belonging to at least one class in the class hierarchy.
(ii) for each such method, the highest class in the specialization hierarchy where it occurs, i.e., the class introducing the method,

(iii) the set of predicates over which the specialization hierarchy is built according to Theorem 2,

(iv) the conjunctive closure of this set of predicates (cf. definition before Theorem 2),

(v) the invariant \( I_{\mathcal{m}} \) of the highest class in the hierarchy, and

(vi) for any class in the specialization hierarchy, the preconditions of the methods belonging to this class.

This information is sufficient to generate any class \( A \) in the specialization hierarchy. Suppose for instance, we want to construct a class \( A \) corresponding to the predicate \( Q = P_1 \land \ldots \land P_k \). Then the according invariant can be derived from (iv) as \( I_{\mathcal{m}} = I \land Q \). The methods (including their signatures, implementations, pre- and postconditions) belonging to \( A \) can be derived from (i), (ii) and (vi); (iii) and (iv) denote the specialization hierarchy, and (v) the information whether \( m \) is a modifier or not, a set of predicates \( P_{\mathcal{m}} \) which specifies the highest class \( A \) in the specialization hierarchy where \( m \) occurs (the minimum condition), an implementation of \( m \) for \( A \) (alternatively, the method could be abstract), a precondition \( \text{Pre}_{\mathcal{m}} \), and a postcondition \( \text{Post}_{\mathcal{m}} \) for \( A \).

For the rest of this section, we use a specification that consists of the following three parts:

(A) a description of all methods (signature and implementation) occurring in some class,

(B) a description from which the conjunctive closure can be derived, and

(C) a description from which invariants, pre- and postconditions can be derived.

This specification contains all the required information:

Obviously, (i) can be derived from (A) and (C). According to Theorem 2, the postconditions of a method \( m \) are identical for all classes in the hierarchy; hence, the necessary postconditions are available from (C). The invariant \( I_{\mathcal{m}} \) of the highest class in the hierarchy can be derived from (C), which satisfies (v).

The equivalence classes of the conjunctive closure are built from a set of predicates \( \text{FRED} \). In addition to \( \text{FRED} \), (B) defines a set of implications and a set of contradictions. An appropriate conjunction of predicates in \( \text{FRED} \) can be used to identify the highest class in the specialization hierarchy containing a method \( m \) in (A), which gives (ii).
CONSTRUCTION OF ROBUST CLASS HIERARCHIES

Example 25: Table XIII shows the predicate table for the domain of directed graphs. Figure 15 provides description (A), specifying some methods belonging to some classes in the robust class hierarchy. If a method \( m \) has return type \( \text{SAME} \), then the return type of this method in a generated class \( A \) is \( A \). The specification assumes a class \( \text{SET} \) with the usual operations and a class \( \text{LIST} \), \( v_1 < v_2 \) denotes that \( v_1 \) precedes \( v_2 \) in a list. In the definition of \( \text{wc} \) which weakly connected components, observe that the result type uses the specialization of weakly connected directed graphs. Hence, it is already specified by the signature, that each component in the result is weakly connected.

The predicate set over which the robust class hierarchy is generated are \( \text{PREDS} = \{ \text{is acyclic}, \text{is tree}, \text{is free} \} \). The implications are \( \text{IMPL} = \{ \text{is acyclic} \Rightarrow \text{is tree} \wedge \text{is free} \} \). There are no contradictions. If we would add a method \( \text{is free} \) which returns \( \text{true} \) iff the directed graph is strongly connected, then the set of contradictions is \( \text{CONTR} = \{ \text{is acyclic} \Rightarrow \text{is free} \} \).

This specification technique suffices to describe the generation of the design alternatives introduced in Section DESIGN OF SPECIALIZATION HIERARCHIES. The following subsections discuss in turn the generation of classes with dynamic properties, non-polymorphic specialization hierarchies, and hierarchies of immutables and factorized classes. In general, specialization hierarchies can be generated from a hierarchy specification \( S = (B, H, P) \) in three major steps:

1. Derive the hierarchy structure \( H \).
2. For every entry in the predicate table (C) different from \( \text{true} \) create an exception.
3. For every class \( C \) of the hierarchy structure \( H \) construct its body and its invariants.

The exceptions created in Step 2 are raised if modifier \( m \) must preserve predicate \( p \), but the formula specified in the corresponding entry is not satisfied.

The hierarchy structure in Step 1 can be created as follows: First, we construct the HASSE diagram of the implication lattice over \( 2^B \). Then we add the edges given by the specified implications. This leads to a set of strongly connected components within the resulting graph. Now, we reduce the graph by folding the strongly connected components into a single vertex. Finally, we remove all transitive edges and associate a name to each remaining vertex. The names can be derived from the predicate names, e.g. \( P_1, P_2, A \), where \( A \) is the base class of the hierarchy.

**Remark:** All formulas in a strongly connected component are equivalent w.r.t. the given specification. This allows us to restrict our consideration to the reduced graph, which is easily

---

Table XIII. Predicate Table.

<table>
<thead>
<tr>
<th>Method</th>
<th>is acyclic</th>
<th>true</th>
<th>is tree</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(v, r)</td>
<td>true</td>
<td>false</td>
<td>is free</td>
<td>true</td>
</tr>
<tr>
<td>remove(v)</td>
<td>true</td>
<td>false</td>
<td>is free</td>
<td>true</td>
</tr>
<tr>
<td>root(v)</td>
<td>true</td>
<td>false</td>
<td>is free</td>
<td>true</td>
</tr>
<tr>
<td>leaf(v)</td>
<td>true</td>
<td>false</td>
<td>is free</td>
<td>true</td>
</tr>
</tbody>
</table>

where \( Q(v_1, v_2) = v_1 \in V \wedge v_2 \in V \wedge v_2 \in V \wedge \text{adj}(v_2, v_1) = 0 \wedge (v_1, v_2) \in E \) and \( \text{R}(X) = X : X \subseteq V \wedge g \wedge E \subseteq \text{E} \).

---

Figure 15. Basic Specification (A) (except).
computed from the strongly connected components in time $O(d^{\text{pred}})$ (cf. 19, Section 5.5).

Example 26: Using the hierarchy specification resulting from Example 25, we can now methodically construct the hierarchy of Figure 2. Figure 16 illustrates the transformation process. The following are the strongly connected components of the Hasse diagram construction:

- $\{\text{isacyclic}\}$
- $\{\text{isacyclic}, \text{isfree}, \text{isacyclic} \land \text{isfree}, \text{isacyclic} \land \text{isfree} \land \text{isfree}\}$

Then, elements of minimal size are chosen, which results in the choice depicted in Fig. 16c.

Generation of Non-Polymorphic Specialization Hierarchies

The generation of a non-polymorphic class hierarchy can be derived from the hierarchy structure computed before, because there is a one-to-one correspondence to the non-polymorphic specialization hierarchy, cf. Theorem 2. Consider the class $P_1 \cdots P_k A$ satisfying $\text{isacyclic} \land \cdots \land \text{isacyclic}$. It can inherit all methods which are not specialized by a stronger precondition from its direct successors in the hierarchy structure. The methods that can be inherited are queries and modifiers which have in the predicate table (B) the entry true or undefined for all columns $\text{isacyclic} \land \cdots \land \text{isacyclic}$. These methods preserve the desired properties $\text{isacyclic} \land \cdots \land \text{isacyclic}$.

The first terms in the conjunction come from the basic specification (A) in Fig. 15, the second terms come from the entry in column $\text{isacyclic}$ of the predicate table (B) in Fig. 16. Furthermore, the class $P_1 \cdots P_k A$ contains additionally all methods of the basic specification (A) with minimum condition $\text{isacyclic} \land \cdots \land \text{isacyclic}$. These are the methods that are only defined if at least $\text{isacyclic} \land \cdots \land \text{isacyclic}$ holds.

Let $m$ be a method that is present in a generalization of class $P_1 \cdots P_k A$ but not inherited from its direct successors. The precondition $\text{Pre}_{m} P_1 \cdots P_k A$ of $m$ is a conjunction of the precondition of the class hierarchy and all entries of the predicate table (C) in row $m$ and column $m$ (an entry undefined is ignored). These are exactly those conditions that are required in order to preserve the stronger invariant. If $m$ is not abstract, its body in the hierarchy specification (A) is wrapped with tests for the preconditions. A method with minimum condition $\text{isacyclic} \land \cdots \land \text{isacyclic}$ in the basic specification (A) has the precondition, postcondition, and body as specified in the basic invariant.

The following steps for the generation of class $P_1 \cdots P_k A$. We summarize the above discussion:

1. The invariant is defined as $\text{isacyclic} \land \cdots \land \text{isacyclic}$.
2. The class inherits from all successors in the hierarchy structure HD (no subtyping)
3. For each modifier $m$ of the basic specification (A) that has a minimum condition $\text{isacyclic} \land \cdots \land \text{isacyclic}$ or in the predicate table (C) an entry in one of the columns $\text{isacyclic}$, $1 \leq i \leq k$ different from undefined and true, introduce a method with the same name and signature. Its precondition is the conjunction of the precondition in the basic specification (A) and the entries different from undefined on row $m$ and column $\text{isacyclic}$ in the predicate table (C). Its postcondition is the postcondition of $m$ in the basic specification (A). Method $m$ is abstract if it is abstract in the basic specification (A). If $m$ is not abstract then its body is

- $\text{Pre}_{m}$ (precondition check)
- $\text{Post}_{m}$ (postcondition check)
- $\text{Exception}_{m}$ (exception raised)

Remark: Step 3 adds the methods that are overridden as well as the methods that are inserted due to the minimum condition $\text{isacyclic} \land \cdots \land \text{isacyclic}$.

Example 27: The class ADG of acyclic directed graphs has according to Step 1 the invariant $\text{isacyclic} \land \text{isacyclic}$. According to Step 2 and the hierarchy HD of Example 26, ADG inherits from the class DG. According to Step 3 the methods make, connect, and sym_add are overridden because there is an entry in column $\text{isacyclic}$ that is different from undefined and true (cf. Table XIII). Their preconditions in class ADG are

- $\text{Pre}_{\text{make}} = \{3y : \text{DG} : y \in V \land \text{sym_add} \equiv \text{EE}\} \land \{3y : \text{DG} : y \in V \land \text{sym_add} \equiv \text{EE}\}$
- $\text{Pre}_{\text{connect}} = \{v : \text{V} \in V \land \text{sym_add} \equiv \text{EE}\}$
- $\text{Pre}_{\text{sym_add}} = \{E \in \text{empty}\}$

The first terms in the conjunction come from the basic specification (A) in Fig. 15, the second terms come from the entry in column $\text{isacyclic}$ of the predicate table (B) in Fig. 16.

Remark: The practical applicability of method sym_add is questionable, because it always raises an exception if the directed graph has an edge. From a practical viewpoint, it might be reasonable to assume that the method sym_add cannot be called for acyclic graphs. If the corresponding entry in Table XIII would be false, then the method would not be present in class ADG.
By slightly modifying the generation algorithm, subhierarchies can be generated from a subset Q of predicates and/or restrictions on the highest class in the hierarchy. Suppose the highest class in a hierarchy must satisfy inv \( \land P_2 \land \cdots \land P_k \). Therefore contains all methods \( m \) of the basic specification \( B \), where \( is.P_1 \land \cdots \land is.P_k \) implies the minimum condition of \( m \). Their pre- and postconditions and their bodies are constructed as above. The hierarchy \( HD \) is constructed using just the set \( Q \) instead of the whole set of predicates. For the combination of both, every vertex in \( HD \) must contain the subset \{ \( P_1, \ldots, P_k \) \}.Figure 17. Class ADG and some of its Methods

Generation of Hierarchies of Immutable Classes

Theorem 4 forms the basis for generating a hierarchy of immutable classes. Since there is one-to-one correspondence between a non-polymorphic specialization hierarchy and a corresponding hierarchy of immutable classes, the overall approach to generating the class hierarchy remains the same, except for the generation of the individual immutable classes. Let \( C \) be a class in a non-polymorphic specialization hierarchy, and \( C^\theta \) denote the corresponding immutable class. The main differences between the non-polymorphic hierarchy and the hierarchy of immutable classes are that

- inheritance also means subtyping,
- every modifier has a return type, and
- for every modifier \( m \), each class introduces a new modifier returning an object of the same class.

Since the classes are conforming according to Theorem 4, inheritance allows for subtyping. The modifiers must be functions due to the immutability of the classes. The last property is necessary, because the returned object may be of any supertype. Additionally, a method \( m \) \( P_1 \cdots P_k \) is created iff in the predicate table (C), each of the entries in columns \( is.P_1, \ldots, is.P_k \) is different from \( true \) and undefined. In this case, the body of \( m \) \( P_1 \cdots P_k \) checks whether the predicates hold before returning. We introduce methods that produce results of a certain explicit class. The preconditions are checked before execution of the body as in the case of non-polymorphic specialization hierarchies. Hence, the generated class \( \Pi.P_1 \cdots P_k.A \) contains for every subset \{ \( Q_1, \ldots, Q_j \) \} \subseteq \{ \( P_1, \ldots, P_k \) \} and every modifier \( m \) a method \( m.Q_1 \cdots Q_j \) with return type \( \Sigma.Q_1 \cdots Q_j.A \). It is only not contained if the modifier need not to strengthen the precondition in order to preserve the stronger invariant than its direct superclasses.

Suppose that the Hasse diagram \( HD \) is already constructed from the basic specification (A), the predicate table (C), and the implications (C). According to the above discussion, the immutable class \( \Pi.P_1 \cdots P_k.A \) is constructed by the following steps:

1. The invariant is \( inv.A \land is.P_1 \land \cdots \land is.P_k \).
2. Inherit conforming from all direct successors of \( HD \).
3. For every method \( m \) specified in the basic specification with minimum condition \( is.P_1 \land \cdots \land is.P_k \) add method \( m \) with
   (a) the signature as specified in the basic specification (A) except that for modifiers the return type is \( \Sigma.P_1 \cdots P_k.A \).
   (b) the precondition of \( m \) is the precondition of \( m \) in the basic specification (A)
   (c) If \( m \) is a query, then the postcondition of \( m \) is the postcondition specified in the basic specification (A).

If the method \( m \) is abstract in the basic specification (A), then the method \( m \) in class \( \Pi.P_1 \cdots P_k.A \) is abstract. If method \( m \) is a query, then the method \( m \) is a method \( m \) in the basic specification (A) and the method \( m \) has the properties of self in the mutable case.

If the method \( m \) is abstract in the basic specification (A), then the method \( m \) in class \( \Pi.P_1 \cdots P_k.A \) is abstract. If method \( m \) is a query, then the method \( m \) is a method \( m \) in the basic specification (A) and the method \( m \) has the properties of self in the mutable case.

Generation of Hierarchies of Immutable Classes

Theorem 4 forms the basis for generating a hierarchy of immutable classes. Since there is one-to-one correspondence between a non-polymorphic specialization hierarchy and a corresponding hierarchy of immutable classes, the overall approach to generating the class hierarchy remains the same, except for the generation of the individual immutable classes. Let \( C \) be a class in a non-polymorphic specialization hierarchy, and \( C^\theta \) denote the corresponding immutable class. The main differences between the non-polymorphic hierarchy and the hierarchy of immutable classes are that

- inheritance also means subtyping,
- every modifier has a return type, and
- for every modifier \( m \), each class introduces a new modifier returning an object of the same class.

Since the classes are conforming according to Theorem 4, inheritance allows for subtyping. The modifiers must be functions due to the immutability of the classes. The last property is necessary, because the returned object may be of any supertype. Additionally, a method \( m \) \( P_1 \cdots P_k \) is created iff in the predicate table (C), each of the entries in columns \( is.P_1, \ldots, is.P_k \) is different from \( true \) and undefined. In this case, the body of \( m \) \( P_1 \cdots P_k \) checks whether the predicates hold before returning. We introduce methods that produce results of a certain explicit class. The preconditions are checked before execution of the body as in the case of non-polymorphic specialization hierarchies. Hence, the generated class \( \Pi.P_1 \cdots P_k.A \) contains for every subset \{ \( Q_1, \ldots, Q_j \) \} \subseteq \{ \( P_1, \ldots, P_k \) \} and every modifier \( m \) a method \( m.Q_1 \cdots Q_j \) with return type \( \Sigma.Q_1 \cdots Q_j.A \). It is only not contained if the modifier need not to strengthen the precondition in order to preserve the stronger invariant than its direct superclasses.

Suppose that the Hasse diagram \( HD \) is already constructed from the basic specification (A), the predicate table (C), and the implications (C). According to the above discussion, the immutable class \( \Pi.P_1 \cdots P_k.A \) is constructed by the following steps:

1. The invariant is \( inv.A \land is.P_1 \land \cdots \land is.P_k \).
2. Inherit conforming from all direct successors of \( HD \).
3. For every method \( m \) specified in the basic specification with minimum condition \( is.P_1 \land \cdots \land is.P_k \) add method \( m \) with
   (a) the signature as specified in the basic specification (A) except that for modifiers the return type is \( \Sigma.P_1 \cdots P_k.A \).
   (b) the precondition of \( m \) is the precondition of \( m \) in the basic specification (A)
   (c) If \( m \) is a query, then the postcondition of \( m \) is the postcondition specified in the basic specification (A).

If the method \( m \) is abstract in the basic specification (A), then the method \( m \) in class \( \Pi.P_1 \cdots P_k.A \) is abstract. If method \( m \) is a query, then the method \( m \) is a method \( m \) in the basic specification (A) and the method \( m \) has the properties of self in the mutable case.

If the method \( m \) is abstract in the basic specification (A), then the method \( m \) in class \( \Pi.P_1 \cdots P_k.A \) is abstract. If method \( m \) is a query, then the method \( m \) is a method \( m \) in the basic specification (A) and the method \( m \) has the properties of self in the mutable case.

Example 28: The generation algorithm generates the immutable class ADG shown in Fig. 18. Some of the methods together that would be generated in the example of the directed graphs have already been shown in Table VII. In particular, the class A contains the methods makeacyclic, addvertext, connectacyclic, deletvertext, deletedge, symglyosacyclic, and transglyos due to Step 4 and method topposet due to Step 3. The methods addvertext, delevertext, and deledge, and transglyos override the corresponding methods of class DG because they preserve the acyclicity. The other methods makeacyclic, connectacyclic, and symglyosacyclic have the same preconditions as the methods make, connect, symglyos, and topposet in the class ADG of the non-polymorphic specialization hierarchy, cf. Example 27.
CONSTRUCTION OF ROBUST CLASS HIERARCHIES

(1) class ADG(VERTEX,EDGE) in the ADG(VERTEX,EDGE)
(2) method make_pyclic(VERTEX,EDGE) in ADG(VERTEX,EDGE) is abstract
(3) method connect_pyclic(VERTEX,EDGE) in ADG(VERTEX,EDGE) is
(4) method vertex_pyclic(VERTEX) in ADG(VERTEX,EDGE) is
(5) method vertex_pyclic(EDGE) in ADG(VERTEX,EDGE) is
(6) method connect_pyclic(VERTEX,EDGE) in
(7) connect_pyclic(VERTEX,EDGE) in
(8) vertex_pyclic(VERTEX) in
(9) vertex_pyclic(EDGE) in
(10) vertex_pyclic(EDGE) in
(11) method del_pyclic(VERTEX) in
(12) method del_pyclic(EDGE) in
(13) method del_pyclic(VERTEX) in
(14) method del_pyclic(EDGE) in
(15) method del_pyclic(VERTEX) in
(16) method del_pyclic(EDGE) in
(17) method del_pyclic(VERTEX) in
(18) method del_pyclic(EDGE) in
(19) method del_pyclic(VERTEX) in
(20) method del_pyclic(EDGE) in
(21) method del_pyclic(VERTEX) in
(22) method del_pyclic(EDGE) in
(23) method del_pyclic(VERTEX) in
(24) method del_pyclic(EDGE) in
(25) method del_pyclic(VERTEX) in
(26) method del_pyclic(EDGE) in

Figure 18. Generated Immutable Class of Acyclic Directed Graphs

The constructor calls of make in the bodies of the basic specification (A) are replaced by make_pyclic, cf. lines (4), (9), (12) and (15). The bodies of method connect_pyclic and sym_class_pyclic raise an exception if the stronger precondition required to preserve acyclicity is not satisfied, cf. lines (7) and (18).

Generation of Hierarchies of Factorized Classes

The generation of factorized classes is similar to the generation of non-polymorphic classes. The generation of the conformance hierarchy of factorized classes is based on Theorem 5. In particular, we can use the same Hasse-diagram as for the non-polymorphic specialization hierarchy. Thus, we compute first the hierarchy of classes HD, then generate the non-polymorphic specialization hierarchy, and finally generate the classes in the factorized hierarchy. Every class in the non-polymorphic specialization hierarchy additionally conforms to the corresponding class of the factorized hierarchy. We denote the factorization of A and all its specializations in the non-polymorphic specialization hierarchy by A. For simplicity, a factorized class A contains only those methods whose preconditions are not strengthened in the specializations of A. The invariant of A is the same as the invariant of A.

Hence, starting from a specialization hierarchy SH, we construct for each class P12...k A of SH a factorized class P12...k A which contains all methods m of P12...k A whose precondition is not strengthened in specializations of P12...k A. This information for a method m can be easily obtained from the predicate table as follows:

1. Let Φ be the precondition of m in the class P12...k A of the non-polynomial specialization hierarchy.
2. Check for all entries Ψ = Q of the predicate table (C) in row m and column Ψ different from P j, i = 1, ..., k whether Φ → Ψ (undeclared is replaced by true). Method m in the factorized class P12...k A iff all these implication are true.

A simple automatic check for the implications in the second step is to check whether Ψ is undefined, true, or contained as a term in Φ (according to Step 3 in the generation of non-polynomial specialization hierarchies, Φ is a conjunction of terms).

If it is decided that a method m belongs to the factorized class P12...k A it has the same precondition, postcondition, and body as method m in class P12...k A of the non-polynomial specialization hierarchy SH.

Example 29: We consider the factorized classes resulting from the one specialization ADG of DG. The factorized class underlineADG contains the method sym_class_p cyclic because the precondition in all specializations of ADG is E.is empty. Furthermore, it contains the query top_sort. It does not contain the method connect because the precondition Pre_connect_ADG does not imply Pre_connect. However, it would be present if the predicate is_free is not considered.

Generating a class with Dynamic Properties

Generating a class for the hierarchy, which uses dynamic properties is done by re-implementing polymorphism in a way such that method dispatch is done depending on the actual state of an object. The basic idea has already been described in subsection Dynamic Properties.

Given a certain hierarchy specification, we generate a class with dynamic properties as follows:

1. For each predicate is P i, generate a private attribute Pi : BOOL. These attributes are used for the decision, which method has to be called.
2. The invariant of the generated class is defined as the invariant of the original base class and the conjunction of the given implications, i.e. Invariant A ∧ Σ i Pi = true ⇒ is P i.
3. For every method m, we generate an according method that
   (a) performs a check, whether the minimal condition is satisfied and
   (b) selects the most special implementation

Assume that the minimum condition for a method is Qi, then there are several special implementations, which can be chosen using the predicates is P i = ... is P j. Then the method may look like in Figure 19.

Figure 19. Implementation Skeleton for Dynamic Properties

Note that the order of the conditions in the implementation selection is very important. If for instance is P i = is P j, then in order to call the most specific method, the check for is P j must be performed prior to the (fall back) check for is P i. This information can be obtained from the set of implications and contradictions (C).

Remark: Every modifier may destroy properties. Therefore, we assume pessimistically that all predicates are destroyed which are not guaranteed to preserved by a modifier m, i.e. the
corresponding boolean attributes are set to false. Only those predicates \( P \) are preserved where the entry at row \( m \) and column \( P \) at the predicate table contains true.

**Example 30:** Fig. 19 shows an excerpt of the class \( D \) of directed graphs with dynamic properties. Its invariant, and the pre- and postconditions of the methods where already discussed in Example 15. Here, we discuss the method implementations of non-abstract methods. \( \text{topsort} \) in the graph example.

**Example 31:** Consider the class hierarchy of sets. The class \( SET(T) \) contains a method \( \text{choose} : T \) with precondition \( \text{Pre}_{\text{choose}}(T) \rightarrow \text{Inv}_{\text{choose}} \) and postcondition \( \text{Post}_{\text{choose}}(T) \rightarrow \text{res} \in \text{self} \). The class of sets over ordered universes, \( \text{ORDERED}_{} \leq \text{ORDERED}(T) \), is a subclass of \( \text{ORDERED}(T) \), and the postcondition could be \( \text{Post}_{\text{ORDERED}}(T) \rightarrow \text{res} \in \text{self} \) if \( \text{res} \leq \text{self} \). The superclasses \( \text{ORDERED} \) and \( \text{ORDERED}_{\text{FINITE}} \) are from Sect. 17. Consider the class \( \text{ORDERED}_{\text{FINITE}} \) which is a class of finite sets.

**Example 32:** Figure 21 shows the basic specification of a hierarchy of classes. The predicate table contains only the entries \( \text{Inv}_1 \rightarrow \text{Inv}_2 \). Let \( \text{Q} \) be a class that conforms to two property classes \( \text{P}_1 \) and \( \text{P}_2 \). Since \( \text{Q} \) is method of \( \text{T} \), and \( \text{T} \) conforms to \( \text{P}_1 \) and \( \text{P}_2 \), both \( \text{Inv}_1 \rightarrow \text{Pre}_{\text{Q}}(\text{T}) \) and \( \text{Inv}_2 \rightarrow \text{Pre}_{\text{Q}}(\text{T}) \) hold. Hence, \( \text{Inv}_1 \rightarrow \text{Pre}_{\text{Q}}(\text{T}) \). Let \( \text{m} \) be a method of \( \text{Q} \). Then either \( \text{m} \) is a method of \( \text{P}_1 \) or it is a method of \( \text{P}_2 \). Consider first the case that \( \text{m} \) is method of \( \text{P}_1 \). Since \( \text{T} \) conforms to \( \text{P}_1 \) and \( \text{P}_2 \), both \( \text{Inv}_1 \rightarrow \text{Pre}_{\text{Q}}(\text{T}) \) and \( \text{Inv}_2 \rightarrow \text{Pre}_{\text{Q}}(\text{T}) \) hold. Hence, \( \text{Inv}_1 \rightarrow \text{Inv}_2 \). By analogy, the same argument holds for methods \( \text{m} \) of \( \text{P}_2 \).

**Conformance Hierarchies**

The generation of conformance hierarchies is based on Theorem 7 and can be handled similar to the generation of non-polymorphic specialization hierarchies. It is, however, based on predicates over generic parameters instead of predicates on objects. We call these predicates properties and assume that specified by abstract classes called property classes which serve as type bounds. This implements the bounded genericity solution from Section GENERICITY.

**Definition 2 (Conformance Hierarchy Specification)** A conformance hierarchy specification is a quadruple \( \mathcal{C} = (\mathcal{B}, \mathcal{Q}, \mathcal{H}, \mathcal{P}) \) where

1. \( \mathcal{Q} \) are the property classes which is a set of generic classes,
2. \( \mathcal{B} \) is the basic specification defined as in Definition 1 except that conjunctions of properties are used (a property is a class \( \mathcal{Q} \) instantiated with generic parameters of \( \mathcal{B} \)),
3. \( \mathcal{H} \) is a hierarchy specification defined analogously to Definition 1 (replacing the set of predicates by a set of properties \( \mathcal{P} \)), and
4. \( \mathcal{P} \) is the predicate table \( \mathcal{P} \) which is a matrix where the rows are indexed by the abstract methods of \( \mathcal{B} \), the columns are indexed by the classes in \( \mathcal{P} \), and the entries contain many-sorted first-order logical formulae.

**Example 32:** Figure 21 shows the basic specification of a hierarchy of classes and Fig. 22 the property classes \( \text{ORDERED} \) and \( \text{FINITE} \). The predicate table contains only the entries true because the postconditions are not strengthened in the subclasses (cf. Example 16). There are no implications between these two properties.

**Remarks:** Property classes define the basis for the restriction of the generic parameters. The set \( \mathcal{P} \) defines the actual basis for the conformance hierarchy. An entry \( \text{false} \rightarrow \mathcal{P} \) defines the supercondition satisfied for \( m \), if the generic parameters satisfy \( \mathcal{Q} \). We allow stronger postconditions just for abstract methods, because otherwise it may require a different implementation to ensure the stronger preconditions.

The conformance hierarchy is generated from a conformance hierarchy specification \( \mathcal{C} \) by the following main steps:

1. Construct the Hasse diagram of conformance hierarchy for the property classes in \( \mathcal{Q} \) from the hierarchy specification \( \mathcal{H} \), i.e. a graph \( \mathcal{HD} = (\mathcal{CLASSES}, \mathcal{CONF}) \) without transitive edges where \( \mathcal{CLASSES} \) is the set of classes (without their body) in a conformance hierarchy, and \( (A, B) \in \mathcal{CONF} \) if \( A \) conforms to \( B \).
2. For every class \( CI \in \mathcal{CLASSES} \) construct its body and its inheritances.
3. Construct from the hierarchy specification \( \mathcal{H} \) the Hasse diagram for the conformance hierarchy, i.e. a graph \( \mathcal{HD} = (\mathcal{CLASSES}, \mathcal{CONF}) \) without transitive edges where \( \mathcal{CLASSES} \) is the set of classes (without their body) in a conformance hierarchy, and \( (A, B) \in \mathcal{CONF} \) if \( A \) conforms to \( B \).
4. For every class \( CI \in \mathcal{CLASSES} \) construct its body and its inheritances.

Steps 1 and 3 are implemented analogously to the generation of non-polymorphic specialization hierarchies. The only difference is that the generic parameters are part of the properties.
CONSTRUCTION OF ROBUST CLASS HIERARCHIES

Example 33: Fig. 22 show the property classes generated by Steps 1 and 2. Fig. 6 show the HASSE diagram for the properties constructed after Step 1 and the HASSE diagram constructed after Step 3.

We consider now Step 4 in more detail. The naming conventions for the class names are analogous to the conventions in the non-polymorphic specialization hierarchy, i.e. $Q_1,\ldots,Q_k\bar{A}$ specifies the class of a hierarchy specification such that the generic parameters satisfy properties $Q_1,\ldots,Q_k$. In the following, we discuss the construction of class $Q_1,\ldots,Q_k\bar{A}$. For simplicity, we assume that the classes in the hierarchy have only one generic parameter. The class $Q_1,\ldots,Q_k\bar{A}$ is generated by the following steps:

1. Generate the class head $Q_1,\ldots,Q_k\bar{A}(T < Q_1,\ldots,Q_k(T))$. Since property classes restrict instances of generic parameters, they are generic. $T$ is the name of the generic parameter as specified in the basic specification $(A)$.

2. Generate subtype inheritances for all direct successors in $HD$.

3. For every abstract method $(T)$ (minimum condition strictly weaker than $Q_1(T) \land \cdots \land Q_k(T)$) of the basic specification $(A)$ where one of the entries of the predicate table $(C)$ in row $m$ and columns $Q_i$, $i = 1,\ldots,k$ (undefined entries are replaced by $true$). The generated method $mT$ is abstract iff it is abstract in the basic specification $(A)$. Otherwise, the generated body is the body of $mT$ in the basic specification $(A)$.

4. For every method $(T)$ of the basic specification $A$ with minimum condition $Q_1(T) \land \cdots \land Q_k(T)$ generate a method $mT$ with the same signature, pre- and postcondition as in the basic specification $(A)$. The generated method $mT$ is abstract iff it is abstract in the basic specification $(A)$.

non-polymorphic specialization hierarchies. It is actually more simple than there, because no exceptions need to be raised.

Figure 22. Implementation of Property Classes

![Figure 22](image-url)
the basic specification (A). Otherwise, the generated body is the body of $m_1$ in the basic specification (A).

Remark: The generalization of the above generating algorithm to more than one generic parameters considers each generic parameter separately. The instance of a generic parameter $T$ is restricted only w.r.t. its properties. E.g., suppose the hierarchy specification $A(U, V)$ with the following properties $Q_1(U), Q_2(V)$ and $Q_3(U, V)$. If all three properties must be satisfied, the class head becomes $Q_1 Q_2 Q_3 A(U < Q_1 Q_3(U, V), V < Q_2 Q_3(U, V))$.

Example 34: Fig. 23 shows the classes of the hierarchy of sets (except the root class) generated by the above algorithm. Consider first the generation of class ORDERED_SET. Step 1 generates the head in line (1) from the head in the basic specification in Fig. 21. Since ORDERED_SET has only one direct successor in the HASSE-Diagram shown in Fig. 6, Step 2 adds the subtype inheritance in line (2). Since no postcondition is strengthened, there are no methods added to Step 3. Step 4 adds the methods $\min, \max, \text{next}$, and $\text{prev}$.

Similarly Step 1 generates the class head (17) for class FINITE_SET and Step 2 generates the subtype inheritance in line (18), because $\text{FINITE_SET}$ has only one direct successor in the HASSE-Diagram shown in Fig. 6. Again, no methods are added due to Step 3 for the same reasons as above. Step 4 adds the method complement.

Finally, consider the generation of class $\text{FINITE\_ORDERED\_SET}$. According to Step 1, the head in line (26) is generated. The class $\text{FINITE\_ORDERED\_SET}$ has two successors in the HASSE diagram shown in Fig. 6. Hence, Step 2 generates the subtype inheritances in lines (27) and (28). Step 3 does not add new methods. Step 4 does not add new methods, because there is no method in the basic specification with minimum condition $\text{FINITE}(T) \land T < T$ (cf. Fig. 21).

 Bridges and Algorithm Classes

The basic principle of bridge generation has already been explained in Subsection Implementation Hierarchies. First, we give the algorithm to generate dynamic bridges, and subsequently, we highlight the differences to the generation of static bridges. The line numbers refer to Figure 9, which illustrates the generation step, considering the graph example. For dynamic bridges an method for copying an abstract object must be provided (cf. lines (3)-(5)).

1. Generate the class header and attributes for references to the representation objects.
2. Further generate the method for exchanging the representation. (lines (1)-(5)).
3. For every derived method $m_1$ generate an attribute $m_2$ that refers to an algorithm class implementing the method (lines (6)-(8)).
4. Generate the initialization methods that initializes the representation attributes and the algorithm attributes (lines (9)-(14)).
5. For every kernel method $m_1$ generate a body that delegates the implementation to the representation class (lines (15)-(23)).
6. For every derived method $m_1$ generate a body that delegates the method to the corresponding algorithm object (lines (24)-(30)).
7. For every derived method $m_1$ generate a default algorithm class $M$, that simply delegates the given method to the representation object. The representation is passed to the algorithm object as a parameter. (lines (32)-(43)).

When generating static bridges (cf. Fig. 11) Steps 1–3 are different: Step 1 creates a class header with an additional generic parameter $\text{REPR}$ (line (1)) bounded by the corresponding abstract class, the representation attribute is of type $\text{REPR}$ (line (2)), and no representation change is added. The last parameter of the classes of the attributes generated by Step 2 are instantiated with $\text{REPR}$ (lines (5)-(7)). Finally, the initializations differ accordingly (lines (8)-(15)).

Summary

This section has demonstrated that specialization, conformance and implementation hierarchies can be generated from rather short specifications. Both construction and maintenance of robust class hierarchies are essentially reduced to the construction of their specification.

The library designer does not need to generate all classes explicitly. Given the results from this Section, library users can, given an easy-to-use tool, interactively select properties, such that only the necessary classes are generated by the tool.

The similar structure of specialization and conformance hierarchy specifications suggests that they can be combined in a straightforward manner.

RELATED WORK

This section discusses several kinds of related work. First, we focus on existing software libraries, ranging from subroutine libraries to object-oriented class libraries, and study how well they satisfy the criteria, and which tradeoffs and priorities were chosen. Second, we highlight other systematic approaches to the construction of robust object-oriented class libraries.

Existing libraries

When the criteria robustness, flexibility, efficiency and safe extensibility are applied at the same time, tradeoffs and priorities must be made. Robustness is obviously an important criterion. Library users cannot be expected to be confident to use the library if unexpected runtime errors may occur and point deeply into the calling hierarchy of the library.
Efficiency is particularly important for libraries of basic algorithms and data structures such as LEDA \(^{20}\). Flexibility and safe extensibility address two different issues: On the one hand, a flexible library is not necessarily safely extensible, since an extension of it may invalidate existing programs by violating invariants. On the other hand safely extensible libraries are not necessarily flexible either.

**Example 35:**

- The FORTRAN libraries BLAS and LAPACK can be easily extended by new routines, but their routines can only be used in exactly the way they are designed for. Therefore, these libraries are not flexible according to our definition.
- The C++ Standard Template Library (STL) \(^{21}\) does not partition the methods of data structure interfaces. Hence, the library does not allow separate exchange of independent implementation aspects. The only exceptions are allocation methods which may be exchanged.
- The Java Class libraries frequently use inheritance and polymorphism, but correct use of the library classes cannot be checked at compile time. Hence, the libraries may not be called robustly in the sense of this article.

The Java collection classes use a pseudo-factorized structure, where the methods within the base classes define some kind of weak contract. E.g. the method for adding an object to a collection normally cannot be specified for any collection in a unique way, since specialized collections restrict the use of that method, by either disallowing duplicates, defining bounds, or some other kind of restriction. Therefore the semantics of method add has been defined as “adds the given object or throws an exception”. This design has one big advantage: All collections may be used in a similar way. But the cost of this artificial similarity is that the user always has to check, whether an operation succeeded or not. Therefore the pure factorization approach seems easy to understand, since the user of a class first has to make sure a data structure implements a method correctly before using it – not afterwards. Another problem is that extensibility is still not guaranteed; the only alternative would be to potentially throw the most general exception in each method. This would force the user to deal with all possible exception states himself within his application, clearly an undesirable alternative.

**Conceptual design of libraries**

The conceptual design of software libraries has been subject to little previous research. We outline the relevant results and compare them to ours.

The generative approach has been suggested by Batory et.al. \(^{1}\). They argue that writing components based on a set of orthogonal properties by hand is inherently non-scalable. As examples, they cite the Booch C++ components, the GNU libg++, NIHCL and COOL. Indeed, none of these libraries seems to be of much practical relevance anymore today. Instead, they propose a generator-based approach similar to ours. Their approach uses a composition technique based on combining layered software components and is not based upon inheritance. Correctness is handled by an external tool that captures semantic information about each component and ensures that only legal systems can be built. Due to the lack of support for any kind of inheritance and polymorphism, the approach cannot be considered flexible. E.g., it is impossible to develop a framework that relies on a certain abstraction and then use polymorphism to replace the expected abstraction with a problem specific implementation.

The use of graphs as a domain to illustrate the design of class libraries is not as wide-spread yet as the use of graphical user interface (GUI) class libraries. It goes back to our original work \(^{21}\). More recently, Pizzonia and Battista \(^{3}\) have also chosen this domain. They introduced a preprocessor-based approach that introduces new language constructs on top of the C++ language. By using “classer” and “extendeer” constructs, they can add additional information and constraints on objects at runtime. Their approach, while being flexible and allowing for the dynamical change of individual properties, has two disadvantages: only dynamic, runtime checks of properties are supported, and implementations of algorithms and data structures cannot be exchanged at runtime.

**Reorganization of class structures**

Opdyke und Johnson \(^{2}\) discuss how to increase the flexibility of class libraries by reducing dependencies between classes. Their strategy is to restructure object-oriented applications by refactoring them. In this sense, their approach is similar to ours, although their goals are limited to the flexibility aspect of our work. Their work describes several basic refactorings, i.e. transformations that retain the original semantics. Factorizing a hierarchy in our sense may benefit from using these and other basic transformations.

Casais \(^{24}\) describes transformations that can be used to reorganize the structure of class hierarchies in order to introduce common abstractions of classes by factorizing similarities such as attributes or methods. Again, the goals are different but the techniques are similar.

**CONCLUSIONS**

Hierarchies of specialized classes are an inherent part of almost any class library. Especially, for libraries of algorithms and data structures, specializations are essential since generally, efficient algorithms require special properties of the underlying data structure. But these specialized classes have one major drawback – their use is restricted with respect to their according base classes. The result is that these classes may not be used polymorphically. Hence a robust library must restrict polymorphic use of specialized classes.

Beside robustness and efficiency, a class library should also provide maximum flexibility. In object-oriented systems, flexibility is usually achieved by polymorphic use of classes. This in turn leads to a vast design problem: How can the library designer provide a class hierarchy that is robust, efficient and allows maximum polymorphic use? In this article, we demonstrated that these properties often contradict. Therefore, we demonstrated various design alternatives for hierarchies of specialized classes. We compared them with respect to the mentioned properties. The comparison indicates that each design alternative has advantages as well as disadvantages. We demonstrated how to leverage the degree of flexibility and efficiency within a robust hierarchy using *static and dynamic bridges*. These allow to separately exchange both representations and method implementations based on certain representations. This is achieved using a combination of the design patterns *bridge* and *strategy*. We demonstrated that robustness can be guaranteed, if either representations or algorithms are restricted.

Although design techniques like the mentioned *bridges* enhance the overall quality of libraries, the core problem still remains: *The* best hierarchy design does not exist in general. But it may exist w.r.t. a certain application context. This leads to our vision of future library architecture (cf. fig. 24). The main idea is to generate hierarchies with certain properties, depending on the application context in which they are used.

We described how implementations of the described design alternatives can be derived from a table based specification. With this technique, the user of a class library may select required classes for the current application and rank the properties flexibility, robustness and efficiency.
Then a class structure that best fits the specified ranking can be automatically generated. This represents an important step on the way to fully customized class libraries, i.e. libraries that may easily be integrated into any application context. With nowadays libraries, this is usually not the case since the class structure of the hierarchy often dominates parts of application development.

ACKNOWLEDGEMENT

We want to thank all the people that helped to improve the presentation of our results. We especially thank all the anonymous referees for their detailed comments on the structure, presentation and formal aspects of the paper. We thank Sabine Glesner for her comments on the paper. Finally, we thank Dorothea Wagner for her help and patience.

REFERENCES

10. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Software Components, Addison-Wesley, 1995.