CS 1999-16

An Evolutionary Approach to Structural Design Composition *

Paulo Alencar, Donald Cowan, Jing Dong, Carlos Lucena
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
{palencar,dcowan,jdong,lucena}@csg.uwaterloo.ca

Abstract

One of the main goals of component-based software engineering (CBSE) is to enable automated assembly of software components, thus bringing greater quality and reliability, reducing development and test times, increasing productivity from multiple use of software components in the software development. A serious impediment for component assembly is due to the fact that the design and architectural assumptions that a reusable component makes about the structure of the application are in most cases implicit, lost or buried in complex implementation structures. In this paper we present an evolutionary approach to support the creation of reusable and changeable software architectures which focuses on the structural composition of components at the design level. The architectural design information, captured by design patterns, is made explicit and represented in a declarative way, being packaged into tangible artifacts as building block design components in the development process. In this way, the declarative design component representations can be instantiated, adapted, assembled, maintained, and implemented. Furthermore, we can also use these representations to reason about properties related to the combination of design components. Our approach is illustrated through a case study involving various design patterns.

Keywords: Design component, design process, design pattern, object-oriented design, Prolog, design reuse, design transformation, software evolution.

1 Introduction

Component-based software engineering (CBSE) focuses on building software systems by assembling prefabricated, configurable, and independently evolving building blocks [19] rather than implementing the entire system from scratch. However, reusable and changeable software architectures, which are amenable to adaptation and composition, are not obtained by a simple combination of component-based applications. A deep knowledge about the domain and about the software design is a critical factor in the construction of such architectures. The apparent lack of design information is one of the most significant barriers that software developers face to reuse or change a component-based software system. Furthermore, the properties required to combine specific components are normally buried in complex implementation structures. Therefore, it is essential for CBSE to explore the combination and the collective behavior of components at the design and architectural levels.

Design patterns, proposed as a means to deal with this issue, capture successful software development design practice within a particular context [12, 6, 8, 22]. They lead to a better understanding about the design assumptions, trade-offs, and implications in a component's implementation. They can be seen as building blocks from which more reusable and changeable software designs can be built. If these building blocks, called design components in [13], are treated as design artifacts the design information embedded in the patterns is not lost in complex implementation structures. In this way, design patterns can be used in a more systematic approach to component-based software development.

*The work described here has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the National Research Council of Brazil (CNPq), and Sybase Inc.

In this paper we present an evolutionary approach to software design composition. The approach is evolutionary because it uses evolutionary transformations, a mechanism to deal with the design changes to be supported and managed throughout the development lifecycle. In our approach, design patterns are packaged into design components using a suitable declarative representation. Using this declarative representation, the design components can be instantiated, adapted, altered, and assembled. Code can also be generated from the design component representations through transformations. Furthermore, our approach also uses a declarative representation of the design component properties. Thus, the design can be checked for inter- or intra-design anomalies. An inter-design anomaly occurs when a design component combination fails to meet some of its required combination properties, while an intra-design anomaly occurs when a design component fails to satisfy one of its internal properties. Finally, the above aspects of our approach are illustrated through a case study.

In the next Section we give a general overview of our approach. Section 3 addresses the issue of checking the consistency of structural composition of design components. In Section 4 we present a case study to illustrate our approach and show how our models help to detect interaction problems or inconsistencies when design components are combined. Finally, Sections 5 and 6 present related work and our conclusions, respectively.

2 The Approach

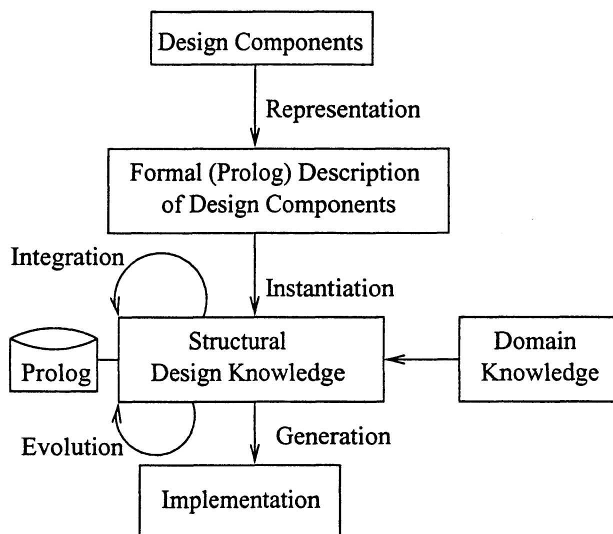

Figure 1: Evolutionary Design Process

Our approach uses a declarative structural representation of the design components and their properties that:

- allows design components to be reused by making the component descriptions available in a component library;
- supports the detection of design component anomalies by providing mechanisms to reason about the individual design components and their interactions;
- supports systematic change of the design component representations through evolutionary transformation techniques.

Figure 1 illustrates the main characteristics of the development process that underlies our approach:

- The representation of design components in a declarative way using Prolog and the storage of these representations in a Prolog database. These design component representations are general or abstract in the sense that they capture good design practice in a domain-independent way;

- The support for the instantiation of the general declarative representations of the design components into concrete domain-specific representations to provide reuse of good design practice;
- The support for the integration or combination of the concrete design components and the intra- and inter-consistency checking about the component composition;
- The support for the evolution of the combined design component representations to incorporate new software system requirements;
- The design component representations in Prolog can be transformed into code templates by tools such as Draco-PUC [16].

It is worth mentioning that intuition and design experience also play an important role in this process as, for example, in the selection of the general design components to be (re)used. In this sense, the information about patterns as described in many pattern catalogues is very useful [12, 8]. Having provided an overview of the evolutionary process, in the next sections we discuss this process in more detail (see Figure 1).

2.1 Representation

As the initial phase of our process, design components, such as design patterns, are represented in Prolog and stored in a Prolog design repository. There are several advantages of using Prolog as our repository or knowledge base. First, the representation of these components can be reused by instantiating the corresponding component Prolog rules each time they are applied to produce a concrete domain-specific component representation. Second, the properties and constraints of each design component can be described as Prolog rules, and these rules can be used to check the intra- and inter-consistency of the design components and their composition, respectively. Third, the addition and removal of structural facts about design components can be accomplished by using the Prolog *assert* and *retract* clauses. Fourth, the transformation of the design component representations in Prolog to code templates can be performed by a transformation tools such as Draco-PUC [16]. Fifth, the design component representations can be recovered through the Prolog deductive facilities [14].

2.1.1 Object-oriented Design Primitives

Design components are represented in terms of object-oriented design primitives in a predicate-like format. Each design primitive consists of two parts: name and argument. The "Name" contains the name of a feature or a relationship in object-oriented design, such as class, inherit, etc. The "Argument" contains general information about a feature or a relation such as the information on the participants of an inheritance relationship. In the following we present the syntax and the meaning of the design primitives we use in this paper:

- class(C): C is a class.
- abstractclass(C): C is an abstract class.
- inherit(A, B): B is a subclass of A.
- attribute(C, A, V, T): V is the name of an attribute in class C with type T. T is optional. A describes the access right of this attribute, i.e. public, private, or protected.
- staticattribute(C, A, V, T): The arguments of *staticattribute* are the same as the ones for *attribute*. The only difference is that we are now defining a static data type.
- method(C, A, F, R, P_1, T_1, P_2, T_2, ...): F is a method of a class C. A describes the access right of this method, i.e. it can be public, private, or protected. R describes the return type. If no return value is required as in the case, for example, of constructors in C++, R can be the value "none". The method's parameters and their types are P_1, T_1, P_2, T_2, ..., respectively, and this part is optional. The return type R is also optional if the method has no parameters.

- staticmethod(C, A, F, R, P_1, T_1, P_2, T_2, ...): The arguments of *staticmethod* are the same as the ones for *method*. The only difference is that *staticmethod* defines a static method (such as a class method in Smalltalk).
- return(C, F, V): V is the return value of the method F in the class C.
- new(C_1, F, C_2, P): This predicate represents a pointer to the dynamic instantiation of class C_2 in the method F of class C_1. P is the initial value of the class C_2. P can contain zero or more parameters depending on the number of parameters the constructor of the class C_2 has.
- assign(C, F, L, R): Right value R is assigned to left variable L in the method F of the class C.
- invoke(C, C_f, O, O_f, P): A method O_f which belongs to the object O is invoked in the method C_f of the class C, where P is the parameter of the method O_f. P can contain zero or more parameters depending on the number of parameters the method O_f has.
- condition(C, F, LC, OP, RC, T, F): This predicate describes the conditional or *if-then-else* statement used by programming languages. It first compares LC and RC with compare operator OP. If the result is true, T is executed. If it is false, F is executed. The conditional statement belongs to method F of class C.
- element(E_1, S_1, E_2, S_2, ...): E_1 is an element of set S_1. E_2 is an element of set S_2, and so on. When universal quantification *forall* and *element* are used together, it enumerates set $S_1, S_2, ..., S_n$ simultaneously, i.e. the first elements of all sets are enumerated first at the same time, then the second elements.

2.1.2 Pattern Primitive Operators

A higher level of abstraction is provided by introducing pattern primitive operators. Pattern primitive operators are represented in terms of design primitive operators and they allow general object-oriented schemas such as delegation, aggregation, and polymorphism to be defined. Pattern primitive operators can capture the sub-patterns which occur frequently in the declarative representation of design patterns as design components. They can also be used to change, transform, or make the pattern declarative representation evolve. This operator can assist with the evolution of the pattern schema and also with the application of this pattern.

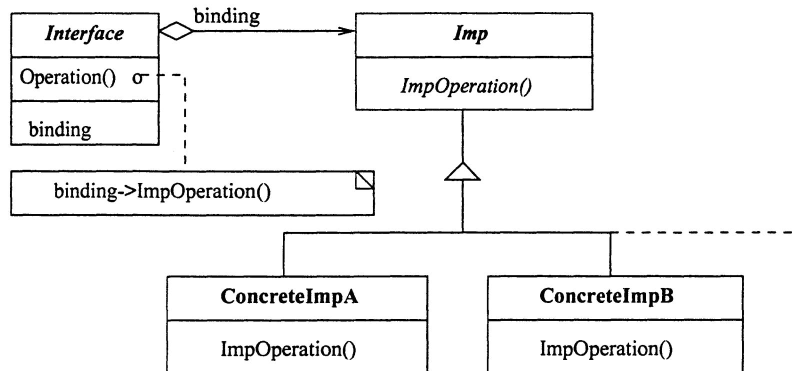

Figure 2: Polymorphism

As an example, a pattern primitive operator called abstract coupling or *polymorph* can be represented in Prolog as follows:

```
polymorph(Interface, Imp, Binding, ConcreteImpSet, ImpOperation, Operation) :-
 assert(abstractclass(Imp)),
 assert(method(Imp, public, ImpOperation)),
 forall(member(ConcreteImp, ConcreteImpSet), assert(inherit(Imp, ConcreteImp)) ),
 forall(member(ConcreteImp, ConcreteImpSet), assert(class(ConcreteImp)) ),
```

```
 forall(member(ConcreteImp, ConcreteImpSet),
        assert(method(ConcreteImp, public, ImpOperation))),
 assert(abstractclass(Interface)),
 assert(attribute(Interface, private, Binding, Imp)),
 assert(method(Interface, public, Operation)),
 assert(invoke(Interface,Operation,Binding,ImpOperation)).

extend_polymorph(Imp,NewConcreteImpSet,ImpOperation) :-
 forall(member(ConcreteImp, NewConcreteImpSet),
       assert(inherit(Imp, ConcreteImp)) ),
 forall(member(ConcreteImp, NewConcreteImpSet),
       assert(class(ConcreteImp)) ),
 forall(member(ConcreteImp, NewConcreteImpSet),
       assert(method(ConcreteImp, public, ImpOperation))).

retract_polymorph(Imp,OldConcreteImpSet,ImpOperation) :-
 forall(member(ConcreteImp, OldConcreteImpSet),
     retract(inherit(Imp, ConcreteImp)) ),
 forall(member(ConcreteImp, OldConcreteImpSet),
     retract(class(ConcreteImp)) ),
 forall(member(ConcreteImp, OldConcreteImpSet),
     retract(method(ConcreteImp,public,ImpOperation))).
```

These *polymorph* rules in Prolog represent the structure shown in Figure 2. The arguments of the *polymorph* predicate denote the generic elements (e.g., class, attribute or method) of the design structure shown in Figure 2. For example, *Interface* and *Imp* are abstract classes. *Binding* represents an object reference which is a state variable of the *Interface* class. *ImpOperation* and *Operation* are two important methods. *ConcreteImpSet* defines a set of concrete classes which includes *ConcreteImpA* and *ConcreteImpB*. This representation contains more information than the OMT representation, which can not, for example, represent an undetermined number of classes. As a side note, this representation can also be seen as a possible solution to the impurity problem of design patterns discussed in [15]. All the arguments will be instantiated by class and operation names and these names result, of course, of specific domain knowledge.

The operators *assert* and *retract* are the Prolog operators used to insert or remove certain facts into or from Prolog database, respectively. The *forall* predicate represents the universal quantification operator. It can quantify over a set of class names and add the corresponding facts about each class name into the Prolog database. For instance, the Prolog rule

```
forall(member(ConcreteImp, ConcreteImpSet), assert(inherit(Imp, ConcreteImp))).
```

corresponds to the following first-order logic formula: $\forall ConcreteImp \in ConcreteImpSet : inherit(Imp, ConcreteImp)$. The non-determinism in *polymorph* leaves space for evolution, i.e., for adding or removing concrete classes which inherit from the abstract class *Imp*. The addition or removal of one such class can be performed by the *extend_polymorph* or the *retract_polymorph* rules, respectively, which in turn *assert* and *retract* the corresponding facts related to the insertion or removal of this concrete class.

Notice that the primitive operators represent basic constituents of an object-oriented design and that the structural information related to design components, such as design patterns, can be represented by pattern primitive operators and design primitive operators. For example, *polymorph* is used to represent the Bridge, State and Strategy patterns as design components. Also, good design practices such as, for example, the Adapter and the *views-a* patterns [5], can also be represented in a declarative way in Prolog using the design primitive operators. In this way, the design of an object-oriented application can be assembled by combining the design components stored in the Prolog database. In addition, the evolution (addition or removal of design components) of a software system design can be achieved by applying specific Prolog rules.

2.1.3 The Bridge Pattern Structural Representation

The representation of the design component related to the structural design information encoded by the Bridge pattern in Prolog is shown as follows and uses the *polymorph* pattern primitive as a design sub-component.

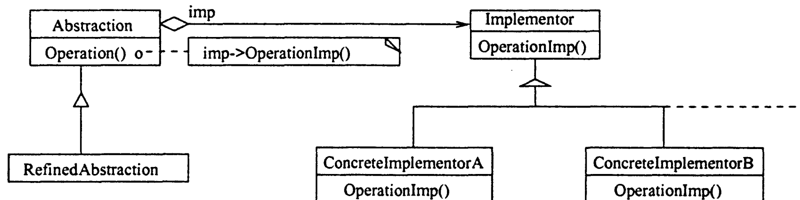

Figure 3: Bridge Pattern

```
bridge(Abstraction,Implementor,Imp,RefinedAbstractionSet,
       ConcreteImplementorSet,ImpOperation,Operation) :-
 polymorph(Abstraction, Implementor, Imp,
        ConcreteImplementorSet,ImpOperation,Operation),
 forall(member(RefinedAbstraction,RefinedAbstractionSet),
        assert(inherit(Abstraction,RefinedAbstraction))),
 forall(member(RefinedAbstraction,RefinedAbstractionSet),
        assert(class(RefinedAbstraction)) ).

extend_bridge_abstract(Abstraction,NewRefinedAbstraction):-
 assert(inherit(Abstraction, NewRefinedAbstraction)),
 assert(class(NewRefinedAbstraction)).

extend_bridge_imp(Implementor, NewConcreteImplementor, ImpOperation) :-
 extend_polymorph(Implementor, NewConcreteImplementor, ImpOperation).

retract_bridge_abstract(Abstraction, OldRefinedAbstraction):-
 retract(inherit(Abstraction, OldRefinedAbstraction)),
 retract(class(OldRefinedAbstraction)).

retract_bridge_imp(Implementor, OldConcreteImplementor, ImpOperation) :-
 retract_polymorph(Implementor, OldConcreteImplementor, ImpOperation).
```

The main purpose of the Bridge pattern is to separate the abstraction from its implementation so that they can vary independently. Its OMT diagram is shown in Figure 3.

Since Bridge pattern allows extensions of both the *refined abstractions* and the *concrete implementations*, there are two rules related to *extend_bridge* and two rules related to *retract_bridge*.

We also provide another example of a Prolog design component representation, which we call the ***views-a*** operator [9, 5], in Appendix A. In [10], we have provided the design component representations of all design patterns in [12].

2.2 Instantiation

When a designer chooses a design pattern to solve a particular application problem during the actual design, he or she can save this design decision as facts in the Prolog database by applying the rules that correspond to the design pattern and using as parameters the domain specific names required by the application. In this way, the Prolog facts which represent the structural constituents of that design pattern component will be saved in Prolog database. This process is similar to the instantiation of classes in object-oriented programming language, but, in

our case, a particular application of the design pattern is being created. Unlike in the case of classes, however, design component instantiation does not involve only structural and behavioral design component information, but also requires documentation to be adjusted and extended to fit the context at hand.

There are four major tasks during instantiation. First, to improve readability and understandability, the generic element names (e.g. classes, attributes, or methods) are replaced by application domain names or domain-specific vocabulary. Each design pattern component encapsulates general design practice that is independent on the application domain. The instantiation of a design pattern component applications leads to domain-dependent designs. This replacement or renaming is achieved by instantiating the arguments of the corresponding design component Prolog rule.

Second, according to the application requirements, a number of concrete components are created. The structural solution provided by design patterns often involves an undefined number of concrete classes, which depends on the application. This undefined character is due to the fact that design patterns are domain-indenpent abstract design solutions. We capture this design pattern characteristic by using sets of elements in Prolog. The arguments in the Prolog rules, which represent design pattern components can be single elements or sets of elements, i.e. one argument may represent a set of elements. Arguments related to sets of elements, when instantiated, assume the value of a fixed number of elements.

Third, design component elements are added as Prolog facts to the design component knowledge base. In the representation of the design pattern component, there is a collection of *assert* statements associated with a Prolog rule. The application of the Prolog rule will automatically lead to insertion of the selected facts into the database through these *assert* statements. Through argument instantiation discussed in previous two tasks, the free variables of the inserted facts are unified with domain-related names.

Fourth, a design pattern component can be extended or retracted if it does not violate the constraints of the component. It is important to have, in the documentation of the design pattern, information about the evolution of the patterns. We also provide rules in Prolog about the evolution of design pattern components. These rules and constraints restrict the addition and removal of elements to avoid undesired interactions among components of a single design pattern. However, these rules do not preclude interactions among different design patterns. We will discuss this issue in Section 3.

2.3 Integration

Integration stands for the assembly of design components into a software system. When a design pattern component is added into Prolog database, this pattern is automatically integrated with all patterns previously stored in the database as long as there is no naming conflict. However, some other issues need to be considered during the integration phase. First, some classes/objects may play different roles when they are the common parts of two different design patterns. We have to make sure that the generic elements in these common parts are instantiated with the same names in the different corresponding Prolog rules. Second, the integration of two or more design patterns may cause undesired interactions among them because they share classes. Some properties or constraints of a design pattern may not hold after integration. In this case, integration is not allowed because of these constraint violations.

2.4 Evolution

Evolution is the activity of adding or removing design elements in existing design. It can happen before or after integration. The addition and removal of system parts should not violate the constraints and properties of design patterns. In the representation of design pattern applications, we provide Prolog rules on extending and retracting design patterns.

Another way of extending an existing design component is through *views-a* operators. As shown in the case study in Section 4.2, a *views-a* operator can add extra functionality to an existing design, thus making it evolve.

3 Reasoning Properties

In this section, we will address the issue of checking the consistency of the representations of the structural design component. We assume an existing object-oriented software system design represented in the OMT or UML notations. Informally, the composition of design components can be achieved by "merging" the graphic representation of each design component. However, there may be inconsistencies among these components. Consistency checking is not an easy task when the graphic notations are used. It requires intuition and experience. The consistency checking is also hard due to the informal notations. On the other hand, representing design components in formal logic notation allows us to describe the properties and constraints of each design component in a precise way and, thus, to automatically check whether a component does not satisfy some properties after it is combined with other components.

There are many reasons to check the consistency of the interactions of a design component assembly. For example, when two components are combined, they may overlap with each other, i.e. they may share some parts. The part that they are sharing can play one role in one component, but another role in the other component. This situation may lead to an inconsistent combination. In addition, an existing design may be modified and gradually evolve. This may lead to a situation in which the new design no longer conforms to the properties that its design components must preserve. The manual discovery of the inconsistency in design can be a difficult job without a formalism or tool support. The design component representations in Prolog allow us to take advantage of the deductive facilities of Prolog to automatically find the inconsistencies. In the next section, we provide a case study to illustrate this process.

4 Case Study

In this section, we provide a case study to illustrate the application of our approach and the discovery of inconsistencies when design components are assembled.

4.1 System Sort

The case study is a simplified variant of a case study presented in [23] which deals with a general-purpose system sort. This application sorts lines of text from standard input and writes the results to standard output. A line is a sequence of characters terminated by a newline. The size of sort files is limited within the main memory. Different sort algorithms, e.g. quick sort, insertion sort, etc., can be chosen at run-time or configured before the system is running. The result will be printed in the order specified by the user. The design of this application is shown in Figure 4 and contains five design patterns: Adapter, Bridge, Factory Method, Iterator, and Strategy.

To address the requirement allowing an interchangeable sorting algorithm, the Strategy pattern was selected to encapsulate the different sorting algorithms, e.g., quick sort, insert sort, selection sort, and etc. In this case, we only deal with comparison-based algorithms. Therefore, all algorithms need a function to compare pairs of elements which can be characters, numbers, file folders, etc. The Bridge pattern captures this abstraction since it decouples the abstraction (comparison) from its implementation (character comparison, number comparison, and folder comparison) so that they can vary independently. The Factory Method pattern defines an interface for creating objects, but lets subclasses (*Char, Num, Folder*) decide which class to instantiate. The Iterator pattern is used to print all sorted elements without exposing its underlying representation. If we have a library containing functions such as, for example, the insert sort, we can reuse some functions required in this design. Since the interface of the insert sort method may not be compatible with the interface of *SortStrategy* method in the *Algorithm* class, we can use the Adapter pattern to adapt the interface.

In [10], we have provided the design component representations in Prolog of the five design patterns previously mentioned. According to the design process shown in Figure 1, we can instantiate each of five design pattern components using the domain knowledge of the system sort in the Prolog database. As represented in Section 2.1, the Bridge pattern structure can be instantiated as *bridge(algorithm, implementor, imp,* {*quick, insert*}*,* {*char, num, folder*}*, compareImp, compare)* in Prolog. Other patterns can be instantiated in a similar way. Therefore, the integration of these five design pattern components can be achieved by instantiating the corresponding Prolog rules and each design pattern component will be stored in the knowledge base as Prolog facts.

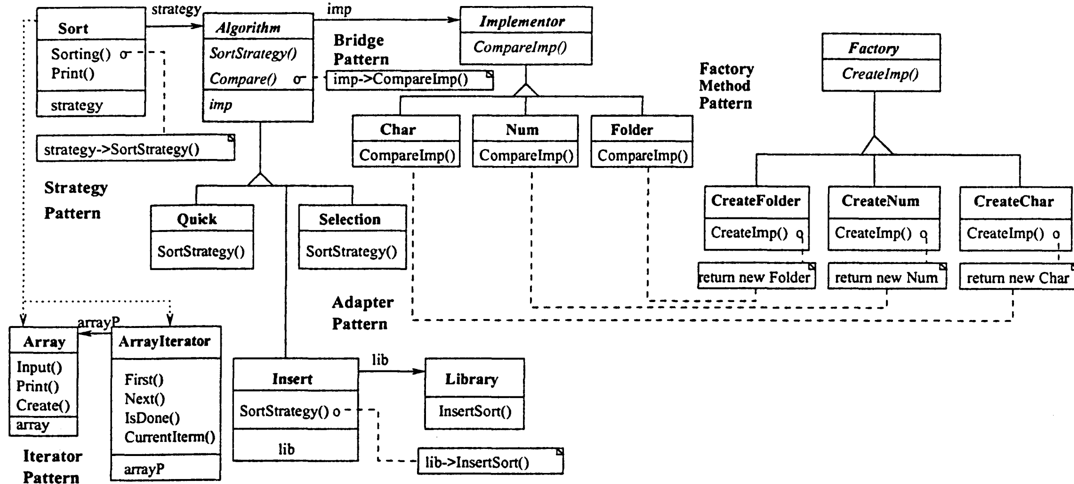

Figure 4: System Sort

From the graphic design representation shown in Figure 4, it is hard to know whether all the properties of each design component are satisfied after the five design components are assembled. It is also not easy to analyze whether there are undesired interactions among design patterns used in this case by just reading the informal design descriptions and the OMT diagrams. In the following, we will show how our formalization allows us to reason about the properties of each design pattern component and find some undesired interactions among the design pattern components in this example.

As shown in the top-left of Figure 4, the application of the Strategy design pattern includes the *Sort, Algorithm, Quick, Insert, Selection* classes and the application of Bridge pattern involves the *Algorithm, Quick, Insert, Selection, Implementor, Char, Num, Folder* classes. In this case, the *Algorithm* class and its descendants are overlapping parts of these two design pattern components. The *Algorithm* class plays the role of *Strategy* in the Strategy pattern and it plays the role of *Abstraction* in the Bridge pattern. The *Insert* class is also an overlapping part of the Strategy pattern, the Bridge pattern and the Adapter pattern. The *Insert* class plays the role of one *ConcreteStrategy* in the Strategy pattern, the role of one *RefinedAbstraction* in the Bridge pattern and the role of an *Adapter* in the Adapter pattern.

Let us analyze the effects of having a design component assembly in which classes assuming different roles in different design pattern components are shared. The main purpose of the Strategy pattern is to encapsulate a family of algorithms and to let the algorithm vary independently of the clients that use it. Therefore, it requires every *ConcreteStrategy* class to overwrite the *SortStrategy* method. This knowledge can be captured in Prolog as a first constraint related to the Strategy pattern: *rule1(Algorithm, SortStrategy) :- inherit(Algorithm, Child), method(Child, public, SortStrategy).*

The Bridge pattern decouples the abstraction (*Compare* method in the *Algorithm* class) from its implementation (*CompareImp* method in the *Implementor* class). Therefore, the *SortStrategy* method uses the *Compare* method to compare pairs of elements without concern about which kinds of elements are compared. This second constraint is a rule related to the Bridge pattern: *rule2(Algorithm, SortStrategy, Compare) :- inherit(Algorithm, Child), method(Child, public, SortStrategy), invoke(Child, SortStrategy, Child, Compare).* There is no inconsistency when we add these two rules to our design knowledge base.

The Adapter pattern adapts the incompatible interface of the *InsertSort* method in *Library* with the *SortStrategy* method in the *Algorithm.* Therefore, the *Insert* class (which has the role of an Adapter) must inherit from the *Algorithm* class (which has the role of a Target) and delegate to the *Library* class (which has the role of an Adaptee). This third constraint is expressed by the rule: *rule3(Algorithm, Insert, SortStrategy, Library):- inherit(Algorithm, Insert), method(Insert, public, SortStrategy), attribute(Library, public, lib, Library), invoke(Insert, SortStrategy, lib, InsertSort).*

In the context of Prolog, when we ask whether our system sort design satisfies all three rules, the Prolog answer

is *no*. This means that there are undesired interactions among these three design pattern components. When we apply both *rule1* and *rule3* no conflict is reported. However, when we try both *rule2* and *rule3*, there is an undesired inconsistency. Thus, the consistency property checking results show us that the composition of the Bridge pattern and Adapter pattern do not satisfy both *rule2* and *rule3*. It happens that after they are combined, neither Bridge pattern nor Adapter pattern preserve the properties they must preserve.

Before we apply Adapter pattern, both *rule1* and *rule2* are satisfied. But, when we apply the Adapter pattern to reuse a library function, the integration of the Adapter pattern and the Bridge pattern violates the constraints of Bridge pattern which separates the abstraction from its implementations and lets them vary independently. For example, if we want to add another type of implementation, e.g. string comparison, for the abstraction of *compare*, we have to worry about the insert sort since the library insert function may not support the implementation of string comparison. As we have discussed in section 2, one of the important qualities of good reusable designs is the ability to evolve. However, when we plug-in the Adapter pattern it impedes the well-behaved evolution of the Bridge pattern.

In our approach, the evolution of design pattern application becomes as simple as adding a series of Prolog facts by applying Prolog rules for extending a design pattern component. The Prolog representations of the design pattern components also contain rules about how to extend and remove some system parts without losing the properties of the design pattern. For example, in Section 2.1 we have included extension and removal rules in the Prolog representation of Bridge pattern design component. If we want to extend the Bridge pattern component to include the implementation of string comparison, we can simply apply the corresponding Prolog rule: *extend_bridge_imp(implementor, string, compareImp)*. Of course, the Factory Method pattern component also needs to be extended since it may be inconsistent with Bridge pattern. This inconsistency can also be checked by our approach.

4.2 Add Views-a Operator

In the previous section, we have demonstrated how to extend the case study through the evolution of design patterns. Let us now introduce another way of making the system evolve during the software maintenance phase.

In Figure 4, an external iterator pattern was applied to print out the sort results without exposing internal structure. The *print* method in *Sort* class can be implemented in C++ as follows:

```
Sort::print(Array *array) {
  ArrayIterator<Array*> i = array->Create();
  for (i.First(); !i.IsDone(); i.Next())
     { i.CurrentIterm()->print(); }
}
```

The *print* method in *Array* class simply prints out the current element. If we the printout in a special format, such as one with an index at the beginning of each element and a semicolon at the end, we need to modify either the *print* method in the *Sort* class or the *print* method in the *Array* class. However, in this case, we need to know both of the implementations of the *print* methods in the two classes. This violates the encapsulation of these classes. In addition, this solution is not a flexible one because we may want other types of printout format.

To achieve black-box reuse of the existing design of this case, we attach a *views-a* operator to the *Array* class. This operator can dynamically add operations to print some information before and after the printout of each sorted element.

As shown in Figure 5, the *Surrogate* class will be attached to the *Array* class in Figure 4 and assume its identity. The *views-a* operator can be added into the design knowledge base by applying the Prolog rule in appendix A as follows: *views-a(view, concreteView1, array, object, print, printStart, printEnd)*. In this way, different printout formats can be achieved by implementing *printStart* and *printEnd* operations in different *ConcreteView* classes without touching the original design and implementation. For example, we can implement the *printStart* method in *ConcreteView1* class by printing the index of current element, and implement the *printEnd* method in *ConcreteView1* class by printing a semicolon. Different printing formats can be combined easily by adding new *ConcreteView* classes and implementing the corresponding *printStart* and *printEnd* methods. The addition and removal of new *ConcreteView* classes can be performed by applying corresponding Prolog rules as shown in Appendix A.

5 Related Work

In previous related work, we have created a formal approach to architectural/design patterns [1, 3]. We have also formally described applications or instances of the patterns and related these applications to (evolutionary) transformations that can be performed in the design phase in a generative way [2, 10]. We have also worked on views [9, 4] and viewpoints as an evolutionary approach to software maintenance [5]. In particular, this result helped us to define a viewpoint-based approach to software system evolution. Further, we have worked on the formal definition of components and their combination [17]. We have also used a higher order logic and its mechanization to check properties about component-based software architectures of user interfaces [3].

As related work, Keller et. al. [13] described a methodical approach to design composition which was illustrated as a process within a four-dimensional design space. Although our approach is also in the area of software composition, it focuses on the evolutionary, declarative, formal, transformational and property-based aspects of design composition.

Earlier work by Pal [21] investigated the law-governed support for realizing design patterns. The author defined some rules and constraints of design patterns. However, in his work the property checking is performed at implementation level. The author did not discuss the interactions among different design patterns when they were integrated together. Other work on tool support for object-oriented patterns [11] also discussed the constraints of patterns. However, they worked on single pattern constraints at implementation level too. Our work emphasizes the interactions among different patterns when they are integrated. The formalizing design patterns [3, 18] can also be seen are related work. We are currently interested in the property checking at design level. Sullivan [25] used Z language to describe the properties of Component Object Model (COM) and discover the inconsistency of COM architecture standard and the mediator style.

In contrast with our previous works on programming understanding [1], software maintenance [5], design assessment [2], system evolution [17], the emphasis of the work described in this paper is on software component reuse at design level. Good design practices are represented as Prolog rules in a database. These good design practices can be reused by instantiation with application domain knowledge. The evolution of the existing design composed by design patterns is restricted by constraints represented in Prolog.

6 Conclusion

In this paper, we have presented an evolutionary approach to design-oriented software composition. The approach focuses on design components, i.e., components that capture architectural design information about the software system. The approach has several advantages. First, it allows design patterns to be represented in a declarative way, thus packaged in tangible design components. Second, our approach provides mechanisms that allow these design components to be reused to form the software application: instantiation, adaptation, assembly, alteration, and generation.

In a general way, the results of this research can impact organizations that are struggling to produce large and complex component-based systems and applications. It is critical for these organizations to have explicit (tangible) component descriptions at the design level instead of having the design and assembly information hidden in complex implementation structures.

We are proposing a more formal approach as current methods for analyzing architectural descriptions are typically ad-hoc, manual efforts with associated high production and maintenance costs. Thus, providing methods to represent and transform design components should assist in minimizing the lack of architectural design information, and the costs of reusing those components. We are convinced that better ways to represent and change design components within the component-based software development approach will significantly improve software engineering efforts.

We have used first-order logic to represent the design components. We have investigated several choices such as Z [24], PVS [20] and Prolog [7]. The syntax of Z is very close to first-order logic representation. However, there is not still enough support for verification in Z. PVS contains theorem prover, which supports automatic reasoning, but it is too large and complex. Prolog supports automatic reasoning and it is compact and easy to learn. The choice of Prolog was compatible with our decision to apply formal methods in a lightweight way to design component assembly.

Our approach has several advantages from the perspective of evolutionary software transformations: (i) it allows for automation (to take care of low-level editing details and to allow changes to be succinctly directed and reliably performed); (ii) it facilitates separation of concerns throughout the development process (the construction of the declarative design component, its instantiation, assembly, adaptation, and code generation are treated as separate activities). (iii) a record of transformation steps preserves the history of how high-level requirements are developed into lower-level specification and how particular combinations take place. This also provides traceability. (iv) the transformation record can be undone and replayed, permitting the exploration of different design choices. (v) design components can be implemented also through transformations.

The approach, discussed in this paper, focuses on structural property preservation and evolution of design composition. We will address behavior properties in the future. Although not all patterns concentrate on structural information, we can also, in principle, use an extension of the approach we have describe in this paper to study the combination of design pattern components which are more behavior than structure-oriented. We are also interested in the effects of the non-functional properties in design component assembly.

A The Views-a Operator

A views-a operator [5], defined as a pattern, was derived from our previous research in Abstract Design View (ADVs) [9]. The views-a pattern is used to compose a new viewpoint with an existing design. It requires several steps. First, the relevant methods in the classes of the viewpoint are divided into predecessors and successors. The views-a pattern is then used to connect the classes in the views to the corresponding classes in the existing system. The views-a pattern model has the structure shown in Figure 5.

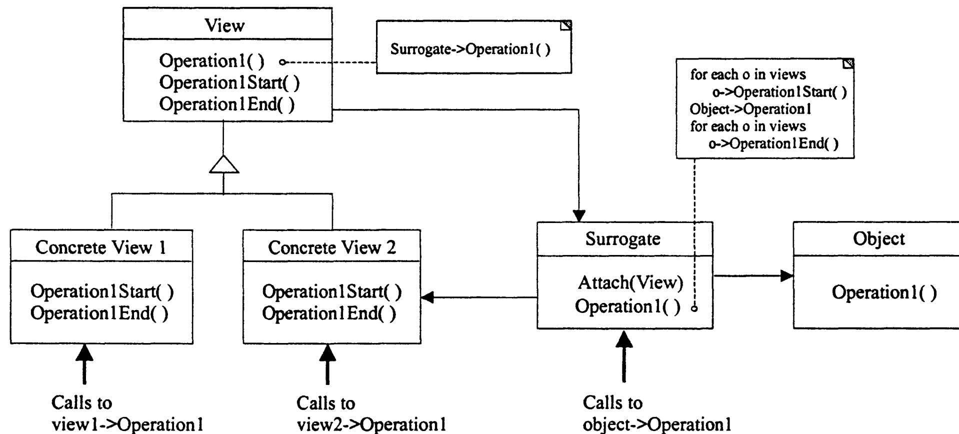

Figure 5: Design model for *views-a* operator

The model works as follows. An object is replaced by a surrogate object, which has the same interface. All views to that object are subclasses of *View* class, which also has the same interface. Each model of a view is divided into two parts, a *Start* method and an *End* method. An *Attach* method in the surrogate allows views to connect to an object dynamically.

When a method is called, it is executed by the surrogate. If it is called in any of the views, the superclass delegates execution to the surrogate. The surrogate calls the appropriate *Start* method from each view, and then the method in the object, and finally the *End* method from each view. The description of the *views-a* pattern in Prolog is as follows:

```
views_a(View, ConcreteViewSet, Surrogate, Object,
  Operation1, Operation1Start, Operation1End) :-

 assert(abstractclass(View)),
```

```
 assert(attribute(View,private,surrogate,Surrogate)),
 assert(method(View,public,Operation1)),
 assert(invoke(View,Operation1,surrogate,Operation1)),
 assert(method(View,public,Operation1Start)),
 assert(method(View,public,Operation1End)),

 forall(element(ConcreteView,ConcreteViewSet),
  assert(class(ConcreteView)),
  assert(inherit(View,ConcreteView)),
  assert(method(ConcreteView,public,Operation1Start)),
  assert(method(ConcreteView,public,Operation1End))
 ),

 assert(class(Surrogate)),
 assert(attribute(Surrogate,private,object,Object)),
 assert(attribute(Surrogate,private,observer,List)),
 assert(method(Surrogate,public,Attach,none,v,View)),
 assert(invoke(Surrogate,Attach,observer,append,v)),
 assert(method(Surrogate,public,Operation1)),
 forall(element(o, observer),
  assert(invoke(Surrogate,Operation1,o,Operation1Start))),
 assert(invoke(Surrogate,Operation1,object,Operation1)),
 forall(element(o, observer),
  assert(invoke(Surrogate,Operation1,o,Operation1End))),

 assert(class(Object)),
 assert(method(Object, public, Operation1)),

 assert(abstractclass(List)),
 assert(method(List, public, append)),
 assert(method(List, public, remove)).

extend_view(View, NewConcreteView, Operation1Start, Operation1End) :-
 assert(class(NewConcreteView)),
 assert(inherit(View, NewConcreteView)),
 assert(method(NewConcreteView,public,Operation1Start)),
 assert(method(NewConcreteView,public,Operation1End)).

retract_view(View, OldConcreteView, Operation1Start, Operation1End) :-
 assert(class(OldConcreteView)),
 assert(inherit(View, OldConcreteView)),
 assert(method(OldConcreteView,public,Operation1Start)),
 assert(method(OldConcreteView,public,Operation1End)).
```

References

[1] P. S. C. Alencar, D. D. Cowan, T. Kuntz, and C. J. P. Lucena. A Formal Architectural Design Patterns-Based Approach to Software Understanding. In *Proceedings of 4th Workshop on Program Comprehension (WPC'96), ICSE-18*, Berlin, March 1996. IEEE Computer Press.

[2] Paulo Alencar, Donald Cowan, Jing Dong, and Carlos Lucena. A Transformational Approach to Structural Design Assessment and Change. *Proceedings of the ECOOP'98 Workshop on the Techniques, Tools and For-*

malisms for Capturing and Assessing Architectural Quality in Object-Oriented Software, Brussels, Belgium, July 1998. LNCS 1543, Springer.

[3] P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena. A Formal Approach to Architectural Design Patterns. *Proceedings of the Third International Symposium of Formal Methods Europe*, pages 576–594, 1996.

[4] P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena. A Logical Theory of Interfaces and Objects. *to appear in IEEE Transactions on Software Engineering*, 1999.

[5] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, and T. Nelson. Viewpoints as an Evolutionary Approach to Software System Maintenance. *Proceedings of the International Conference on Software Maintenance, Bari, Italy*, 1997.

[6] F. Buschmann, R. Meunier, H. Rhonert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture.* John Wiley and Sons, 1996.

[7] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Berlin : Springer-Verlag, 1987.

[8] James Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design.* Addison-Wesley, 1995 and 1996.

[9] D.D. Cowan and C.J.P. Lucena. Abstract Data Views: An Interface Specification Concept to Enhance Design. *IEEE Transactions on Software Engineering, SE-21(3)*, pages 229–243, March 1995.

[10] Jing Dong. A Transformational Process-Based Approach to Object-Oriented Design. *Master's Thesis, Computer Science Department, University of Waterloo*, 1997.

[11] G. Florijn, M. Meijers, and P. van Winsen. Tool Support for Object-Oriented Patterns. *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, pages 472–495, June 1997.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[13] Rudolf K. Keller and Reinhard Schauer. Design Components: Towards Software Composition at the Design Level. *Proceedings of the 20th International Conference on Software Engineering*, pages 302–311, 1998.

[14] Christian Kràmer and Lutz Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. *Proceeding of the Working Conference on Reverse Engineering, IEEE CS press, Monterey*, November 1996.

[15] A. Lauder and S. Kent. Precise Visual Specification of Design Patterns. *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, pages 114–134, July 1998.

[16] J.C.S.P. Leite, M. Sant'Anna, and F.G. Freitas. Draco-PUC: A Technology Assembly for Domain Oriented Software Development. *Proceedings of the third IEEE International Conference of Software Reuse*, November 1994.

[17] C.J.P. Lucena and P.S.C. Alencar. A Formal Description of Evolving Software Systems Architectures. *Science of Computer Programming, 24(1)*, pages 41–61, 1995.

[18] Tommi Mikkonen. Formalizing Design Pattern. *Proceedings of the 20th International Conference on Software Engineering*, pages 115–124, 1998.

[19] Oscar Nierstrasz and Laurent Dami. Component-Oriented Software Technology. *Object-Oriented Software Composition, ed. O. Nierstrasz and D. Tsichritzis, Prentice Hall*, pages 3–28, 1995.

[20] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering, 21(2)*, pages 107–125, Feb. 1995.

[21] Partha pratim Pal. Law-Governed Support for Realizing Design Patterns. *Technology of Object-Oriented Languages and Systems (TOOLS), USA*, pages 25–34, July 1995.

[22] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1994.

[23] D. Schmidt. Object-Oriented Design Patterns and C++ – Advanced C++ Features and Design Patterns. *Lecture Notes, Washington University*, pages 1–122, 1996.

[24] J.M. Spivey. The Z Notation, A Reference Manual. *Prentice Hall*, 1992.

[25] Kevin J. Sullivan, John Socha, and Mark Marchukov. Using Formal Method to Reason about Architectural Standards. *Proceedings of the 19th International Conference on Software Engineering*, pages 503–513, May 1997.