CS 2000-20

The Verification of Hypermedia Design Composition*

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

Abstract

Developing large-scale software systems by integrating software components becomes important practice due to the increasing complexity and size of these software applications. However, the unexpected interactions among the components of software systems are often the cause of failure of component-based systems. The analysis of the properties of the components and their compositions allows us to detect these interactions and correct the composition errors. Discovering composition errors early in the development process can save considerable effort of fixing them downstream. This paper introduces a rigorous analysis approach based on model checking to software design composition. In particular, the analysis goal is to verify whether properties related to structural, behavioral and evolutionary aspects of the design component specifications hold when these components are integrated. We illustrate our approach with a hypermedia case study, showing how to represent, instantiate and integrate design components, and how to find design composition errors using model checking techniques.

Keywords: Software design and analysis, complex software systems, component-based software development, formal methods, web-based systems, model checking, hypermedia design patterns, design components.

1 INTRODUCTION

Software product information becomes more frequently delivered as hypermedia documents because of the availability of the World-Wide Web and its associated communications infrastructure. As one of the world's largest publishers of printed and machine-readable documentation, IBM is constantly seeking new ways to create, manage and deliver documentation to make it more effective for the customer and less expensive to produce and manage. This joint industry/university project is motivated by the difficulties that large software companies face when they have to create, manage and maintain large size of rapidly changing complex hypermedia documents of product information. LivePage [10] was proposed to address the documentation needs of IBM within this new Web and Internet-based world.

However, although developing hypermedia systems with component-based techniques [25] can lead to fast time-to-market, complex software applications such as these normally cannot be built from simply putting software components together. Many experiments [12, 14] indicated that deep knowledge about the domain and about the software design is a critical factor in the construction of such applications. Many software components must guarantee critical functional, fault-tolerant, real-time, and performance properties. Proving such properties still hold after the composition of these components can increase our confidence on the correctness and reliability of the integration. Proofs based on formal, rather than informal, techniques make our reasoning precise; moreover, they are amenable to mechanical aids such as syntax and semantics checkers. Errors in the designs of these systems are difficult and expensive to find and correct if propagated to the implementation phase. Designers must be able to formally analyze the composition of the software components at the design level.

*This work is being supported in part by the NSERC, and the CSER grant funded by IBM Canada and NRC.

Design components [16] have been proposed to reify good design practice, such as design patterns [11], from conceptual design building blocks into a tangible and composable form. Design components focus on achieving component-based problem solving instead of component-based implementation. People are especially interested in discovering and documenting such reusable design experience in different domains as, for example, in hypertext design [24]. However, less work has been done on reasoning about the composition of these design patterns.

There is an increasing interest in modeling software by various formalisms, and checking properties or finding errors against the models by model checkers [6]. Numerous examples can be found [7] in areas such as requirement analysis, distributed cache coherence analysis, word processor design analysis, mobile IP protocol analysis, CAD algorithm analysis, real-time operating system kernel analysis, and Java meta-locking algorithm analysis.

In this paper, we describe a generic specification approach for design components in hypermedia applications and a framework in which the composition of such components can be verified. The approach allows us to characterize structural and behavioral aspects, and a specific form of evolution of these components. The main contributions of this paper can be summarized as follows: (i) an approach to specify an abstract design component in a generic way. This generic specification can be instantiated to the model of a concrete design component with the corresponding domain knowledge of the application. The specification language is based on a well-known declarative programming language, Prolog, which reduces the user's learning curve; (ii) a process that supports the verification of design components in order to analyze through model checking whether composition properties hold; (iii) the definition of some aspects (structural, behavioral and evolutionary) of the design analysis with the specification language that allows us to check not only the properties of each aspect independently, but also to check how changes to one aspect may affect properties of other aspects. For example, we are able to check whether behavioral properties of the composition hold when the system structure changes, or whether the structural and behavioral properties still hold when the design component evolves by the addition or removal of a component element; (iv) a case study applying the systematic approach to specify and verify the composition of design components in the hypermedia domain.

The remainder of this paper is organized as follows. Section 2 describes the gist of our analysis techniques including a design analysis process based on abstract design component specifications and model checking techniques. Section 3 gives details of a case study illustrating the design analysis process. We analyze a hypermedia design by checking whether relevant structural, behavioral and evolutionary system properties hold. The last two sections are about related work and conclusions.

2 ANALYSIS TECHNIQUE

In this section, we introduce our design analysis process including design component representation, instantiation, integration and property checking. We also describe the model checking technique used in our design analysis to verify composition properties.

Figure 1 illustrates the main characteristics of the design analysis process underlying our approach. Design components are *represented* in a declarative way using XL. The representations are generic in the sense that they capture good design practice in a domain-independent way. These declarative representations, which constitute models of the design components, are *instantiated* into concrete domain-specific representations and, in this way, design practice can be reused. The concrete design components are *integrated* to be a model of the design composition, Σ, which is then subjected to automated verification.

2.1 Formal Specification and Verification Techniques

Formal automated verification techniques constitute the core of our analysis approach. Essentially, we rely upon Prolog proofs [27] and model checking techniques [6] to analyze the composition and integration of design components. Model checking entails comparing two formal objects (Σ, ϕ), e.g. the software design components and their compositions are portrayed as a logic model, Σ, and the properties of these components are represented as logic formulas, ϕ. One assumes that if a formula ϕ is true in the model Σ, then the corresponding property holds in the model of the design. We use a model checker as a black box to check Σ against the property specification, ϕ. The model checker outputs either true, if Σ satisfies ϕ, or a counterexample, if it does not. When a property

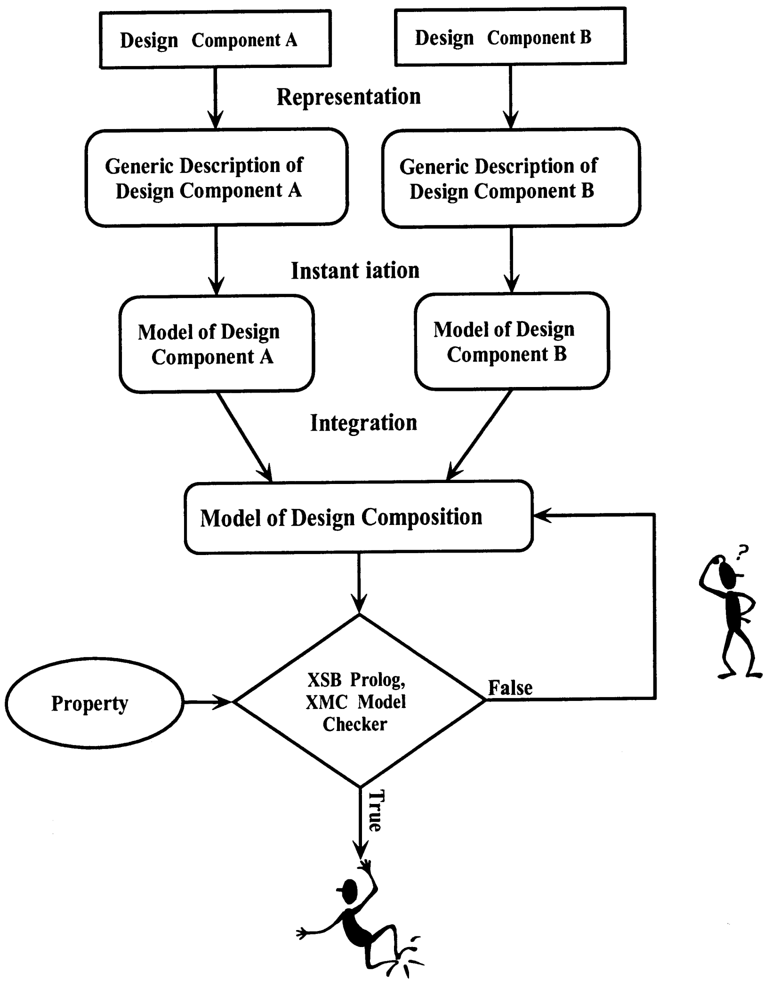
Design Component A
Design Component B
Representation
Generic Description of Design Component A
Generic Description of Design Component B
Instant iation
Model of Design Component A
Model of Design Component B
Integration
Model of Design Composition
Property
XSB Prolog, XMC Model Checker
False
True
?

Figure 1: The Design Analysis Process

violation is found, we can go back to check and update the design composition. One reason why this verification technique is so promising is that model checking can be automated for many temporal logics.

XMC [21] is a model checker for verifying temporal properties of a system. It is written in the XSB tabled Prolog programming system [27]. Temporal properties are expressed in the alternation-free fragment of the μ-calculus, a very expressive temporal logic [18]; the system to be verified is described in the model specification language for XMC (called XL) which is a highly expressive extension of value-passing CCS [20]. Prolog terms and predicates are used respectively to represent values and computations. Thus specifications can make use of recursive data structures and computations. XMC has been successfully used to verify various systems as documented in [22].

In the next subsections we describe each phase of our design analysis approach. In particular, we show how the structural and evolutionary design component representations are specified in XSB Prolog by rules and facts and the behavioral design component representations are specified in XL, the XMC specification language. In this way, we will be able to verify structural and evolutionary composition properties using the XSB Prolog deductive facilities and verify behavioral composition properties expressed in the μ-calculus temporal logic using the XMC model checker.

2.2 Representation

In the initial phase, design components, such as design patterns, are represented in Prolog and XL and stored in a XSB Prolog database. There are several advantages of using XSB Prolog as a repository of design knowledge. First, the representations of these components can be reused by instantiating the corresponding generic Prolog and XL descriptions of each component when it is applied to produce a concrete domain-specific component representation. Second, the properties and constraints of each design component can be described in temporal logic and thus be checked in XMC. Third, the addition and removal of structural facts about design components can be accomplished by using the Prolog *assert* and *retract* clauses. Fourth, design components can be recovered through Prolog deductive facilities.

2.2.1 Structure

The structural representation of design components is specified in XSB Prolog in terms of object-oriented design primitives in a predicate-like format. Each design primitive consists of two parts: *name* and *argument*. The *name* part contains the name of an entity or a relation in object-oriented design, such as class, inheritance, etc. The *argument* part contains generic information about an entity or a relation such as the information on the participants of an inheritance relation. In the following, we present the syntax and the meaning of the design primitives used in this paper[1]:

- class(C): C is a class.
- abstractclass(C): C is an abstract class.
- inherit(A, B): B is a subclass of A.
- variable(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, e.g. public, private, or protected.
- 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, e.g. public, private, or protected. R describes the return type. If no return value is required R can be the value "void". The method's parameters and their types are P_1, T_1, P_2, T_2, ..., respectively, and are optional. The return type R is also optional if the method has no parameters.
- 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.
- 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.

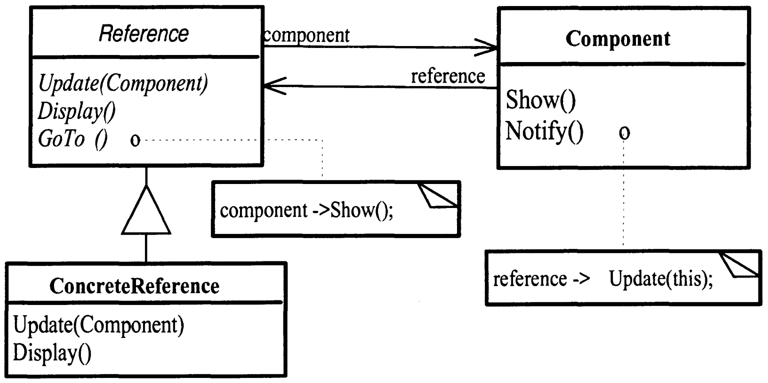

Figure 2: Active Reference Pattern (Class Diagram)

As an example, we use the Active Reference pattern [24] to illustrate our representation formalism. In many hypermedia applications, particularly those with spatial or time structures, the user usually needs to have visual knowledge about the current location in terms of spatial or time space during the navigation. Providing this information not only helps the user to find out the current position in the complex navigation space, but also allows him or her to change to other position freely. The Active Reference pattern was proposed to address this issue by providing a perceivable and permanent reference about the current status of navigation. The current status is usually highlighted. One common example of the application of the Active Reference pattern is to keep an index permanently visible on the screen while navigating a multi-page document. An OMT description of this pattern is shown in Figure 2. The *Component* class is the navigation component, in which the *Show* operation is defined to show its contents on the screen. The *Notify* operation is used to notify the change of the current navigation status as, for example, closing the display of the current component and opening another component. The *Reference* class is an abstract class which defines the interface of a list of operations. The *Update* operation is used to change the visual highlight showing the current position in the navigation structure when a new navigation component is on display. The *Display* operation is to display or refresh the active reference on the screen. The *GoTo* operation is defined to change the current status by directly selecting an item on the active reference to display the corresponding component. The *ConcreteReference* class implements different concrete active references. For instance, an index can be a textual active reference to a document; a map can be a graphic active reference to a travel information system.

The representation of the structural aspect of the Active Reference pattern in XSB Prolog is shown as follows:

```
active_reference(Component,Reference,ConcreteReferenceSet,
          Show, Notify, Update, Display, GoTo) :-
 assert(class(Component)),
 assert(method(Component, public, Show)),
 assert(method(Component, public, Notify)),
 assert(variable(Component,private,reference,Reference)),
 assert(invoke(Component, Notify, reference, this)),
 assert(abstractclass(Reference)),
 assert(method(Reference, public, Update, void, component, Component)),
 assert(method(Reference, public, Display)),
 assert(method(Reference, public, GoTo)),
 assert(invoke(Reference, GoTo, component, Show)),
 forall(member(ConcreteReference, ConcreteReferenceSet), assert(class(ConcreteReference))),
 forall(member(ConcreteReference, ConcreteReferenceSet),
     assert(method(ConcreteReference, public, Update, void, component, Component))),
 forall(member(ConcreteReference, ConcreteReferenceSet),
```

[1]A more comprehensive set of design primitives is described in [8, 2].

```
    assert(method(ConcreteReference, public, Display))).
```

The Prolog rule, `active_reference`, represents the structural aspect of the Active Reference pattern. The arguments of `active_reference` denote the generic elements, e.g., classes, attributes, or methods. For example, `Reference` is an abstract class; `Show` is a method. The Prolog operators, `assert` and `retract`, are used to insert or remove certain facts into or from the Prolog database, respectively. The `forall` predicate represents the universal quantification operator. When it is used with the `member` predicate, it can quantify over a set of class names and apply a Prolog rule on the selected members.

2.2.2 Behavior

The behavioral design component representations are specified in XL, the model specification language for XMC. XL is a highly expressive extension of value-passing CCS.

The syntax of XL specification is given as follows:

```
Pdef ->  ( Pname ::= Pexp . )*
Pname -> Term
Pexp -> Pexp o Pexp      Prefix
  | Pexp # Pexp          Choice
  | Pexp '|' Pexp        Parallel Composition
  | Pexp @ PortMap       Relabelling
  | Pexp \ PortList      Restriction
  | Pname                Recursion
  | in(Port,Term)        Communication (input)
  | out(Port,Term)       Communication (output)
  | action(Term)         Communication (non-sync)
  | Comp                 Computation (Prolog expression)
  | if(Comp, Pexp, Pexp) Conditional Expression
  | zero                 Empty process (0 in CCS)
  | nil                  Empty computation
PortMap -> [Port / Port (, Port / Port)*]
PortList -> { Port (, Port)* }
Term -> PrologTerm
Comp -> PrologPredicate
Port -> PrologAtom
```

`Pname` is a parameterized process name, represented as a Prolog term; `Comp` is a computation, e.g., X is Y+1. Process `in(Port,Term)` inputs a value over port `Port` and unifies it with term `Term`; `out(Port,Term)` outputs term `Term` over port `Port`; Process `action(Term)` specifies an action that is represented by `Term` and used for non-synchronous communication. Process `if (Comp, Pexp, Pexp)` behaves like the first `Pexp` if computation `Comp` succeeds and otherwise like the second `Pexp`. Operation 'o' is sequential composition; '|' is parallel composition; '#' is nondeterministic choice; '@' is relabeling where `PortMap` is a list of substitutions; and '\' is restriction where `PortList` is a list of port names. Recursion is provided by a set of process definitions, `Pdef`, of the form `Pname ::= Pexp.`

Consider, for example, the specification of the Alternating Bit Protocol [26] in XL. We assume that any text after the % character is a comment.

```
medium(Get, Put) ::=
    in(Get, Data);
    {   out(Put, Data)
    #   action(drop)
    };
    medium(Get, Put).
```

```
sender(AckIn, DataOut, Seq) ::=
    % Seq is the sequence number of
    % the next frame to be sent
    out(DataOut, Seq);
    {
        in(AckIn, AckSeq);
        if AckSeq == Seq
            %% successful ack, next message
            then {
                NSeq is 1-Seq;
                sendnew(AckIn, DataOut, NSeq)
            }
            %% unexpected ack, resend message
            else sender(AckIn, DataOut, Seq)
    #
        %% upon timeout, resend message
        sender(AckIn, DataOut, Seq)
    }.

sendnew(AckIn, DataOut, Seq) ::=
    action(sendnew);
    sender(AckIn, DataOut, Seq).

receiver(DataIn, AckOut, Seq) ::=
    %% Seq is the expected next sequence number
    in(DataIn, RecSeq);
    if RecSeq == Seq
        then {
            NSeq is 1-Seq;
            action(recv);
            out(AckOut, RecSeq);
            receiver(DataIn, AckOut, NSeq)
        }
        else {
            %% unexpected seq, resend ack
            out(AckOut, RecSeq);
            receiver(DataIn, AckOut, Seq)
        }.

abp ::=
      sendnew(R2S_out, S2R_in, 0)
    | medium(S2R_in, S2R_out)       % sender -> receiver
    | medium(R2S_in, R2S_out)       % receiver -> sender
    | receiver(S2R_out, R2S_in, 0).
```

The process `medium` represents a noisy channel. The `sender` process sends a packet to the channel and waits for an acknowledgement. Upon timeout, it resends the packet. The `receiver` process receives a packet from the channel and sends an acknowledgement back. The `abp` process is the parallel composition of the previously described processes.

The UML [5] collaboration diagram shown in Figure 3 describes the dynamic aspect of the Active Reference pattern, which contains four messages between two objects. The message r_1.`GoTo` and c_2.`Show` present a sequence of operations with the subscript of the name of each object instance defines the time order of these two operations.

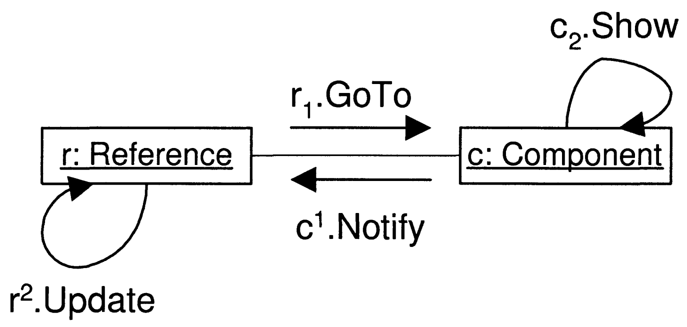

Figure 3: Active Reference Pattern (Collaboration Diagram)

Similarly, c^1`.Notify` and r^2`.Update` presents a time order by the superscript of the name of each object instance. Note, Figure 3 describes two separate sequences which may not correlate together. Superscripts and subscripts are used to separate these two sequences. It is also possible to use two collaboration diagrams to make this distinction.

The behavioral aspect of the Active Reference pattern is modeled in terms of the collaboration among the methods in the classes of this pattern in XL shown as follows:

```
rGoTo(Reference, GoTo, Component, Show, C) ::=
     out(C, (Reference, GoTo, Component, Show))
   o action(r_goTo(Reference, GoTo)).

cShow(Reference, GoTo, Component, Show, C) ::=
     in(C, (Reference, GoTo, Component, Show))
   o action(c_Show(Component, Show)).

cNotify(Component,Notify,Reference,Update, R) ::=
     out(R, (Component, Notify, Reference, Update))
   o action(c_Notify(Component, Notify)).

rUpdate(Component, Notify, Reference, Update, R) ::=
     in(R, (Component, Notify, Reference, Update))
   o action(r_Update(Reference, Update)).

ref(Reference,Component,GoTo,Update,Show,Notify) ::=
     rGoTo(Reference, GoTo, Component, Show, C)
   | cShow(Reference, GoTo, Component, Show, C)
   | cNotify(Component,Notify,Reference,Update, R)
   | rUpdate(Component, Notify, Reference, Update, R).
```

where each XL process describes the behavior of a method and the `ref` process defines the behavior of this pattern as the parallel composition of these processes. Each message shown as a line with an arrowhead in Figure 3 represents a communication between two objects. One object sends the message to a channel and the other object receives the message from this channel. Each object may perform some actions before or after the message is sent. For example, rGoTo (r_1`.GoTo`) defines a process that sends the message `(Reference, GoTo, Component, Show)` to a channel `C` and performs an action. `cShow` (c_2`.Show`) is a process that receives the message `(Reference, GoTo, Component, Show)` through the channel `C` and also performs an action.

2.2.3 Evolution

The evolutionary representation of design components is also specified in XSB Prolog in terms of object-oriented design primitives in a predicate-like format. For example, the non-determinism in `active_reference` leaves space for evolution, i.e., for adding or removing concrete classes which inherit from the abstract class `Reference`. The addition or removal of such classes can be performed by the `extend_reference` and `retract_reference` rules respectively, which in turn `assert` or `retract` the corresponding facts related to the insertion or removal of these concrete classes.

```
extend_reference(Component, Reference, NewConcreteReference, Update, Display, GoTo) :-
 assert(class(NewConcreteReference)),
 assert(inherit(Reference, NewConcreteReference)),
 assert(method(NewConcreteReference, public, Update, void, component, Component)),
 assert(method(NewConcreteReference, public, Display)).

retract_reference(Component, Reference, OldConcreteReference, Update, Display, GoTo) :-
 retract(class(OldConcreteReference)),
 retract(inherit(Reference, OldConcreteReference)),
 retract(method(OldConcreteReference, public, Update, void, component, Component)),
 retract(method(OldConcreteReference, public, Display)).
```

2.3 Instantiation and Integration

Whenever a component is used in a specific application, it needs to be instantiated to include the application domain information. This process can be achieved by unifying the arguments of the description of each design component with terms representing domain information. On the other hand, the composition of two design components can be achieved by overlapping their common parts.

In summary, the design components are represented as Prolog theories (first-order theories), but our results can be applied to other formalisms. Component instantiation is based on theory interpretation, a formal approach to refinement. Composition is also based on theory interpretation and on some criteria to ensure that the composition of design components is correct. However, in this paper, instead of focusing on our instantiation and composition techniques, we concentrate on the verification of composition properties. For a more detailed description of the composition techniques see, for example, [9].

2.4 Property Checking

As we have previously mentioned, we will verify structural and evolutionary composition properties using the XSB Prolog deductive facilities and verify behavioral composition properties expressed in the μ-calculus temporal logic using the XMC model checker.

The μ-calculus temporal logic is a modal calculus whose semantics is usually described over sets of states of labeled transition systems. The μ-calculus is encoded in XMC in an equation form as follows:

```
D ->   Z += F (least fixed point)
     | Z -= F (greatest fixed point)
F -> Z | tt | ff | F \/ F | F /\ F | <A> F | [A] F
```

`Z` is a set of formula variables encoded as Prolog atoms; `A` is a set of actions; `tt` and `ff` are propositional constants; $\wedge$ and $\vee$ are standard logical connectives; `<A> F` (`[A] F`) denotes that possibly (necessarily) after the action `A` the formula `F` holds.

Some temporal properties, such as deadlock and drop package, are described in μ-calculus as follows:

```
%% The system can deadlock.
deadlock += [-] ff \/ <-> deadlock.
```

```
%% A packet can be lost without being received
drop_packet += <sendnew>lost \/ <->drop_packet.
lost    += <sendnew>tt \/ <-recv>lost.
```

These properties can be checked against the model of the Alternating Bit Protocol by XMC.

3 CASE STUDY

In this section, we first describe two hypermedia design components and then analyze their compositions by representation, instantiation and integration of these components. Properties are checked against the structural and behavioral aspects of the composition model. We also check whether structural and behavioral properties hold when a design component evolves by the addition or removal of a component element. We have adopted the case study related to the design of the LivePage system [10].

3.1 Two Design Components

Hypermedia design patterns [24] have been proposed to reuse design experience, to improve communication within and across software development teams, to capture explicitly the design decisions made by designers, and to record design tradeoffs and design alternatives in hypermedia applications. A number of design patterns have been discovered, such as the Navigation Observer pattern, the Active Reference pattern, the Navigational Contexts pattern, and the Information on Demand pattern. A comprehensive catalog of hypermedia design patterns can be found in [13]. In the following, we will use the Active Reference pattern and the Navigational Contexts pattern, as an example, to show the description of these design pattern components and their composition, and to verify the properties by a model checker (XMC).

3.1.1 Active Reference

The Active Reference pattern was previously described in Section 2.2.

3.1.2 Navigational Contexts

Hypermedia applications usually involve navigating collections of nodes, which may be explored in different ways according to the task the user is performing. For example, collections of paintings may be studied author by author, or explored by different categories, e.g., nature paintings or architecture paintings. The Navigational Contexts pattern [24] separates the context information from the content of a hypermedia component and dynamically attaches different context information to a component. This enrichment of the navigation interface, when a component is visited in that context, can be achieved similar to the Decorator pattern [11] (see the OMT diagram in Figure 4). If the collections of hypermedia components are modeled as an aggregate similar to that in the Composite pattern [11] (see the OMT diagram in Figure 5), the Navigational Contexts pattern can be seen as the integration of the Decorator pattern and the Composite pattern. Its OMT diagram is shown in Figure 6. The names of some classes and operations have been changed to represent the corresponding meanings in the Navigational Contexts pattern, e.g., the *Decorator* class is changed to the *Context* class, the *Leaf* class is changed to the *Content* class, and all *Operation* operations are changed to the *Show* operations which are used to display the corresponding content or context information on the screen.

3.2 Representation

The representation of the structural, behavioral and evolutionary aspects of the Active Reference pattern was previously described in Section 2.2.

The Prolog description of the structural aspect of the Navigational Contexts pattern can be achieved by incorporating all corresponding name changes (discussed in the previous section) and instantiating the Prolog descriptions of the *Decorator* pattern and the *Composite* pattern[2] as follows:

[2]The Prolog description of the structural aspect of the Composite pattern can be found in [8].

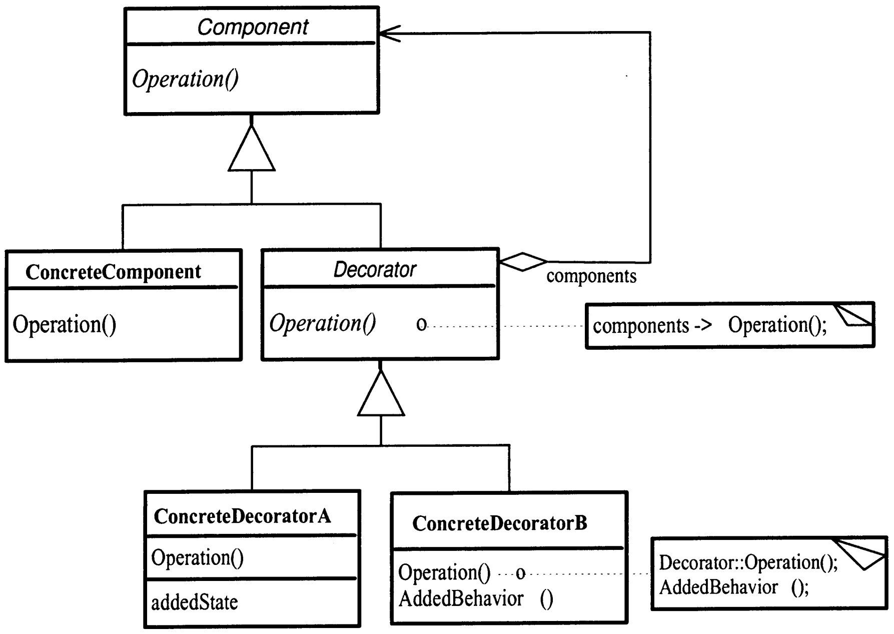

Figure 4: Decorator Pattern (Class Diagram)

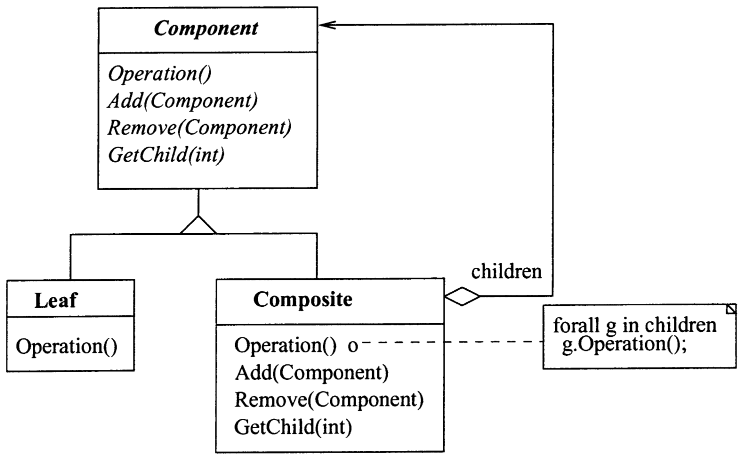

Figure 5: Composite Pattern (Class Diagram)

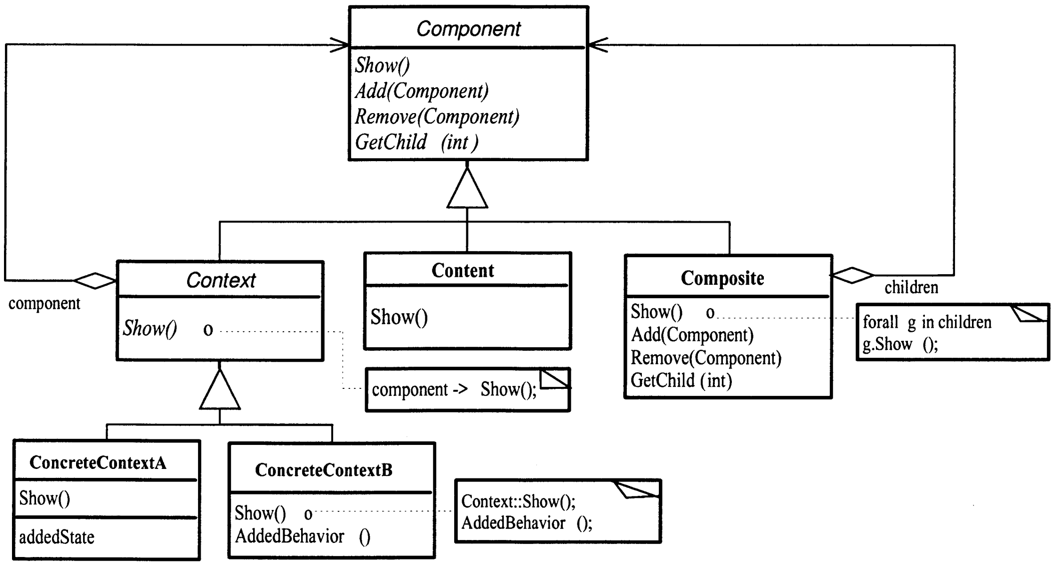

Figure 6: Navigational Contexts Pattern (Class Diagram)

```
navigational_contexts(Component, Context, ConcreteContextSet, Content, Composite,
       Show, Add, Remove, GetChild, AddedBehavior, Components, Children)  :-
  decorator(Component,Content,Context,ConcreteContextSet, Show,AddedBehavior,Components),
  composite(Component,Composite,Content,Children,Show).
```

where the decorator rule describes the structural representation of the Decorator pattern:

```
decorator(Component, ConcreteComponent, Decorator, ConcreteDecoratorSet, Operation,
          AddBehavior, Components) :-
 assert(abstractclass(Component)),
 assert(method(Component, public, Operation)),
 assert(inherit(Component, ConcreteComponent)),
 assert(class(ConcreteComponent)),
 assert(method(ConcreteComponent, public, Operation)),
 assert(inherit(Component, Decorator)),
 assert(abstractclass(Decorator)),
 assert(variable(Decorator, private, Components, Component)),
 assert(method(Decorator, public, Operation)),
 assert(invoke(Decorator, Operation, Components, Operation)),
 forall(member(ConcreteDecorator, ConcreteDecoratorSet),
   assert(inherit(Decorator, ConcreteDecorator))),
 forall(member(ConcreteDecorator, ConcreteDecoratorSet),
   assert(class(ConcreteDecorator) )),
 forall(member(ConcreteDecorator, ConcreteDecoratorSet),
   assert(method(ConcreteDecorator, public, Operation))),
 forall(member(ConcreteDecorator, ConcreteDecoratorSet),
   assert(method(ConcreteDecorator, public, AddBehavior))),
 forall(member(ConcreteDecorator, ConcreteDecoratorSet),
   assert(invoke(ConcreteDecorator, Operation, Decorator, Operation))).
 forall(member(ConcreteDecorator, ConcreteDecoratorSet),
   assert(invoke(ConcreteDecorator, Operation, ConcreteDecorator, AddBehavior))).
```

The evolutionary aspect of the Navigational Contexts pattern is represented as follows:

```
extend_contexts(Context, NewConcreteContext, AddedBehavior, Show) :-
  extend_decorator(Context, NewConcreteContext, AddedBehavior, Show).

retract_contexts(Context, OldConcreteContext, AddedBehavior, Show) :-
  retract_decorator(Context, OldConcreteContext, AddedBehavior, Show).

extend_content(Component, NewContent, Show) :-
  extend_component(Component, NewContent, Show).

retract_content(Component, OldContent, Show) :-
  retract_component(Component, OldContent, Show).
```

where we have reused the following evolutionary representation of the Decorator pattern:

```
extend_decorator(Decorator, NewConcreteDecorator, AddBehavior, Operation) :-
 assert(inherit(Decorator, NewConcreteDecorator)),
 assert(class(NewConcreteDecorator)),
 assert(method(NewConcreteDecorator, public, Operation)),
 assert(method(NewConcreteDecorator, public, AddBehavior)),
 assert(invoke(NewConcreteDecorator, Operation, Decorator, Operation)).
 assert(invoke(NewConcreteDecorator, Operation, ConcreteDecorator, AddBehavior)).

extend_component(Component, NewConcreteComponent, Operation) :-
 assert(inherit(Component, NewConcreteComponent)),
 assert(class(NewConcreteComponent)),
 assert(method(NewConcreteComponent, public, Operation)).

retract_decorator(Decorator, OldConcreteDecorator, AddBehavior, Operation) :-
 retract(inherit(Decorator, OldConcreteDecorator)),
 retract(class(OldConcreteDecorator)),
 retract(method(OldConcreteDecorator, public, Operation)),
 retract(method(OldConcreteDecorator, public, AddBehavior)),
 retract(invoke(OldConcreteDecorator, Operation, Decorator, Operation)).
 retract(invoke(OldConcreteDecorator, Operation, ConcreteDecorator, AddBehavior)).
retract_component(Component, OldConcreteComponent, Operation) :-
 retract(inherit(Component, OldConcreteComponent)),
 retract(class(OldConcreteComponent)),
 retract(method(OldConcreteComponent, public, Operation)).
```

The behavioral aspect of the Navigational Contexts pattern is modeled in terms of the collaboration (see Figure 7) among the methods in the classes of this pattern in XL:

```
xShow(Context, Component, Concrete, Show, R, T) ::=
    in(R, (Concrete, Show, Context, Show))
  o out(T, (Context, Show, Component, Show))
  o action(x_Show(Context, Show)).

cShow(Context, Component, Concrete, Content, Composite, Show, P, T) ::=
  { in(P, (Composite, Show, Component, Show))
  # in(T, (Context, Show, Component, Show)) }
  o
  { contextShow(Context, Component, Concrete, Show)
  # contentShow(Content, Show)
  # compositeShow(Composite, Component, Show)
```

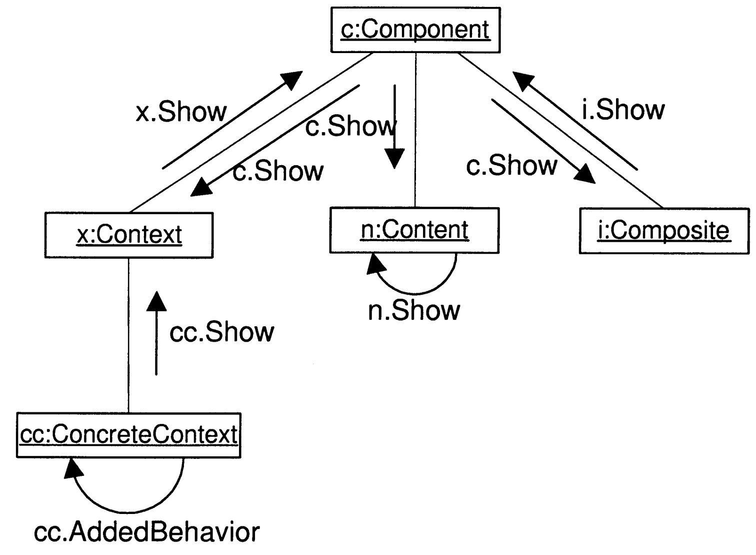

Figure 7: Navigational Contexts Pattern (Collaboration Diagram)

```
  }
  o compShow(Context, Component, Concrete, Content,
             Composite, Show).

nShow(Content, Show) ::=
    action(n_Show(Content, Show)).

iShow(Composite, Component, Show, P) ::=
    out(P, (Composite, Show, Component, Show))
  o action(i_Show(Composite, Show)).

ccShow(Concrete, Context, Show, R) ::=
    out(R, (Concrete, Show, Context, Show))
  o action(cc_Show(Concrete, Show))
  o action(AddedBehavior).

nav(Context, Component, Concrete, Content, Composite, Show) ::=
    xShow(Context, Component, Concrete, Show, R, T)
  | cShow(Context, Component, Concrete, Content, Composite, Show, P, T)
  | nShow(Content, Show)
  | iShow(Composite, Component, Show, P)
  | ccShow(Concrete, Context, Show, R).
```

3.3 Instantiation

In the previous section, we have shown the generic description of each pattern in XL. Whenever a component is used in a specific application, it needs to be instantiated to include the application domain information. This process can be achieved by unifying the arguments of the description of each design pattern component with terms representing domain information. For instance, the Active Reference pattern can be instantiated as the design of a collection of paintings in a museum with a map as an active reference showing the current visiting location by highlighting it on the map. This instantiation process is shown as following: *active_reference(painting, ref_interface, [map], show, notify, update, display, goTo).* The behavioral aspect can be instantiated as following: *ref(ref_interface, painting, goTo, update, show, notify).* Therefore, the design decision and information of the Active Reference pattern are written in the XSB Prolog database which can be composed with those of other component instances.

For the example of exploring the paintings in a museum, the user may need to study them with different contexts through context links. This design decision can be realized and recorded by the application of the Navigational Contexts pattern as: *navigational_context(painting, context, [button], content, composite, show, add, remove, getChild, showButtons).* The behavioral aspect can be instantiated as: *nav(context, [button], painting, content, composite, show).* The context link can be a list of buttons, e.g., first, next, previous, last buttons, which are used to connect all hypermedia components with the same kind of context by a linked list for easy navigation. For example, Van Gogh's painting *Sun Flowers* can be reached while exploring paintings about nature, where the context information can be the list of buttons connecting to the next, previous, first, or last paintings about nature in this collection. On the other hand, *Sun Flowers* can be accessed as a Van Gogh's work, where the context information can be the list of buttons connecting to the next, previous, first, or last paintings of Van Gogh's work.

The instantiation of the generic design components is similar to framework instantiation, where the classes keep their relationships (e.g. inheritance and association relations). Only the names of the classes and their attributes and operations are changed to include the application domain information. Classes in each concrete design component can be further instantiated to objects using conventional object-oriented techniques. For example, Van Gogh's painting *SunFlowers* can be an instance of the class *painting* in the concrete Navigational Contexts component. This instance is represented as $\boxed{\underline{\text{Sun Flowers:painting}}}$ in both the object and the collaboration diagrams.

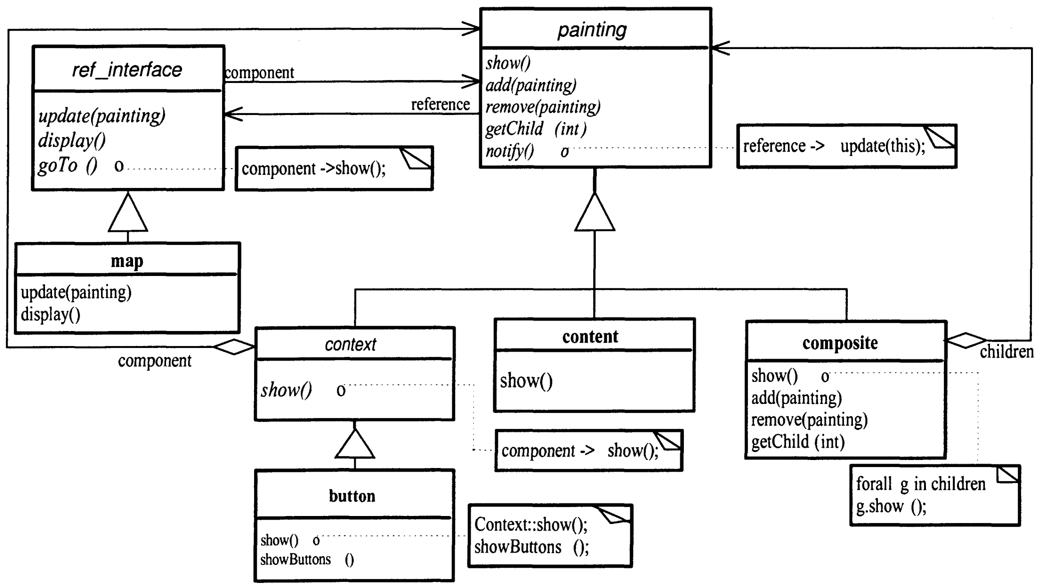

Figure 8: The Design Composition

3.4 Integration

As the application requires both having an active map showing the current position of the user in a museum and being able to explore the museum according to different contexts, we can compose the two design component instances described in the previous section to achieve these goals. The composition can be achieved by overlapping their common parts. For example, as shown in Figure 8, the *painting* class is an overlapping part of the two design components (see Figure 2 and Figure 6). The integrated design can be described as a process in XL (Figure 9). Essentially, this design, which we call `reference_context`, is the parallel composition of the structural and behavioral models of each of the two design components.

```
reference_context ::=
    active_reference(painting, ref_interface, [map], show, notify, display, goTo)
    | navigational_context(painting, context, [button], content, composite, show,
        add, remove, getChild, showButtons)
    | ref(ref_interface, painting, goTo, update, show, notify)
    | nav(context, [button], painting, content, composite, show).
```

Figure 9: The Integration in XL

3.5 Design Analysis Results

The goal of our design analysis is to be able to find design composition errors, with the help of model checking tools, which can be difficult to detect by visual inspection. In this section, we describe the discovery and correction of two design errors in the design composition: one error is related to the structural aspect and the other is related to the behavioral aspect. We then show how to check the behavioral properties with the structural evolution of the design.

There are many different structural properties that can be verified in this application. A simple example related to the structural consistency is that a class cannot be defined as both abstract and concrete. This property is a simple syntactic property of the UML object model and is related to basic type checking of the structural representations. The negation of this property is represented as: `inconsistent -= class(A) /\ abstractclass(A)`. The verification result showed there was class definition inconsistency because the *painting* class was defined as both an abstract class and a concrete class in the composition of the Prolog descriptions. The reason for this inconsistency is that the Active Reference pattern defines the *Component* class, whose instance is the *painting* class, as a concrete class whereas the Navigational Contexts pattern defines the *Component* class, whose instance is the *painting* class, as an abstract class. This means that the *Component* class in the Active Reference pattern cannot be overlapped with the *Component* class in the Navigational Contexts pattern and instantiated to the *painting* class. Instead, it should overlap with the concrete component (the *content* class) in the Navigational Contexts pattern. Thus, the *ref_interface* class in Figure 8 (an instance of the *Reference* class in the Active Reference pattern in Figure 2) should have association relationships with the *content* class instead of with the *painting* class. This change can be simply achieved in our design composition model by updating the bold underlined parts from `painting` to `content` in Figure 9.

After the composition takes place and some classes are identified, we can also check properties that do not amount to basic type checking. As an example, we can check whether classes that have been identified in the resulting composition are still correctly connected to its dependent classes. The following property describes that the design composition does not affect the inheritance relationships in the Navigational Contexts pattern component:

```
forall(member(inherit(A,B), navigational_contexts(painting, context,
            [button], content, composite, show, add, remove, getChild, showButtons)),
       member(inherit(A, B), reference_context)).
```

As we continue our analysis on the new design composition with the help of XMC, we found out another error of the new design. The idea of the Active Reference pattern is to have a permanent and visible reference to a navigation structure and be able to change the current position by calling the *goTo* operation in the *ref_interface* class. Therefore, the invocation of the *goTo* operation should eventually invoke both the *show* operation in the *content* class to display the content of a hypermedia component (`liveness1`), and the *show* operation in the concrete context class, i.e. the *button* class, to display the context information of the hypermedia component (`liveness2`). These two liveness properties are described generically in XMC as follows:

```
========================================================
liveness1(Reference, GoTo, Content, Show) -=
    [r_GoTo(Reference, GoTo)] formula1(Content, Show)
  /\ [-] liveness1(Reference, GoTo, Content, Show).

formula1(Content, Show) +=
     <n_Show(Content, Show)> tt
  \/ form1(Content, Show)
  \/ [-] formula1(Content, Show).

form1(Content, Show) +=
     <n_Show(Content, Show)> tt
  \/ [-{r_GoTo(_,_)}] form1(Content, Show).

========================================================
liveness2(Reference, GoTo, ConcreteContext, Show) -=
    [r_GoTo(Reference, GoTo)]
    formula2(ConcreteContext, Show)
  /\ [-] liveness2(Reference,GoTo,ConcreteContext,Show).

formula2(ConcreteContext, Show) +=
     <cc_Show(ConcreteContext, Show)> tt
  \/ form2(ConcreteContext, Show)
  \/ [-] formula2(ConcreteContext, Show).

form2(ConcreteContext, Show) +=
     <cc_Show(ConcreteContext, Show)> tt
  \/ [-{r_GoTo(_,_)}] form2(ConcreteContext, Show).
```

These descriptions can also be instantiated to represent the liveness properties in this application as follows: `liveness1( ref_interface, goTo, content, show). liveness2 ( ref_interface, goTo, button, show).` The model checking of these liveness properties shows that the first liveness property, that the *show* operation in the *content* class is eventually invoked, holds. However, the second liveness property, that the *show* operation in the concrete context class (the *button* class) is eventually invoked, does not hold. Therefore, when the user clicks on the active reference (e.g. the map of a museum) to change the current position, only the content of the newly chosen component will be displayed. The context information (the buttons) of this component will not be shown. We have lost all context information and are not able to navigate by the context links. The solution to this problem is to move the overlapping part further down to the concrete context class (button). This change can be achieved in our design composition model by updating the bold underlined parts from `painting` to `button` in Figure 9. The model checking results show that both liveness properties hold this time.

One of the advantages of using design patterns is that they cope with the evolution of the designs. We encode this evolution information in the descriptions of each design component by the `extend_` and `retract_` rules. When the application requires to have another kind of context information, e.g. *text* information, in addition to *button* information, we can achieve this design decision of structural change simply by instantiating a predefined Prolog rule (see Section 3.2) as follows: `extend_contexts(context, text, showText, show)`. For example, Van Gogh's painting *Sun Flowers* can be reached while exploring paintings about nature through a context link of buttons, additionally, it can contain the information about the natural aspect of the painting *Sun Flowers* as another kind of context information. On the other hand, *Sun Flowers* can be accessed as a Van Gogh's work. In addition to a context link of buttons connecting Van Gogh's work, the new context (*text*) can contain the information about the relationships with other paintings by him. The modified design is shown in Figure 10. It contains one additional concrete context class (*text*). Therefore, we need to ensure the *show* operation in this new concrete context class will be eventually invoked when the *goTo* operation in the *ref_interface* class is called. This property can be described by instantiating the second liveness property as: `liveness2(ref_interface, goTo, text, show)`. The model checking results show that this property holds in the modified design. Because the evolution of the design composition may affect the properties related to the unchanged parts as shown in [1], we need to make sure other properties we have checked still hold after the design changes. For this case, we have also verified the (`liveness1`) and the `inconsistent` properties in the modified design and there was no reported error this time.

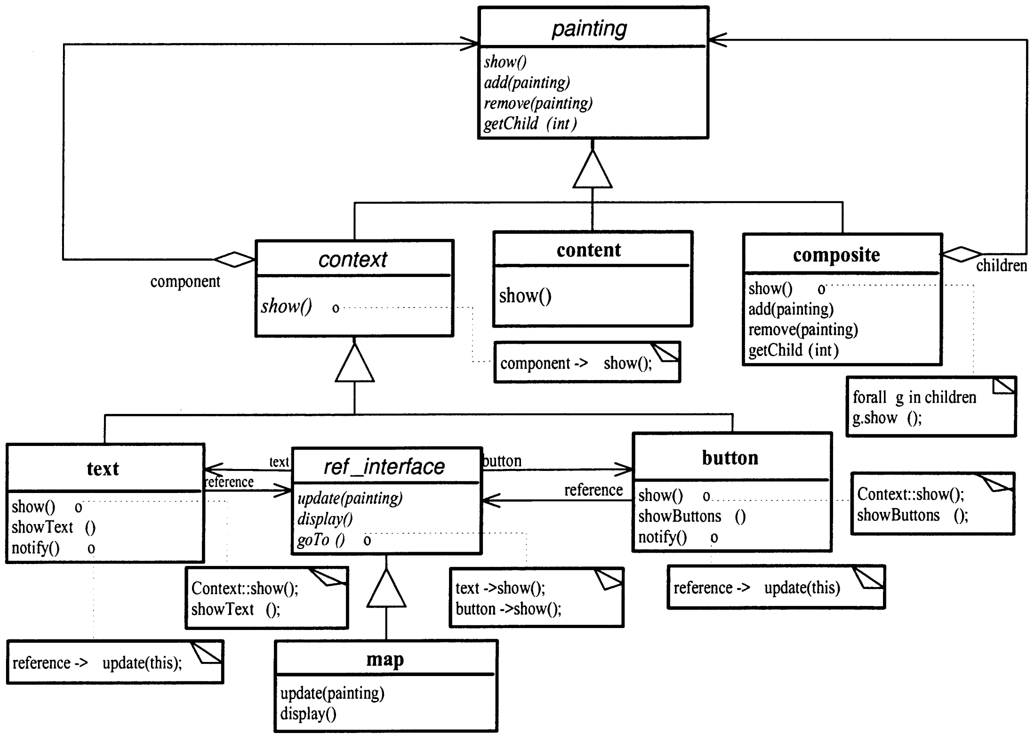

Figure 10: The Evolved Design Composition

4 RELATED WORK

Keller et al. [16] 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 formal, declarative, and property-based aspects of design composition.

Batory et al. [4] provided domain-independent algorithms to validate component compositions for the GenVoca model of software generators. In addition to syntactical checking, such as type checking, they provided design rule (domain-specific constraint) checking to ensure semantical correctness. The design rule checking was achieved by the debugging capabilities of a general utility based on attribute grammars. In contrast, our work focuses on reasoning about the design compositions.

Riehle [23] proposed an analysis method for the composition of design patterns. Role diagrams were introduced to describe the patterns, and a role relation matrix was used to visually depict the composition constraints. His work was restricted to deal with patterns based on object collaborations, and lacked generality and formal treatment of composition.

Various design pattern recovery methods and tools [17, 15] are related to our work in the sense that all recovered patterns can be modeled and checked for anomalies in their compositions. The correction of these anomalies completes the reengineering tasks.

Formalizing design patterns and architecture patterns has been proposed in [3, 19]. Although Mikkonen [19] has discussed the composition of two design patterns based on a formal method, his approach relies on a specific specification language (DisCo). The correctness depends on the refinement correctness of this language since the composition is achieved in terms of refinement. Our approach emphasizes on specifying design components and their compositions, and checking the properties by a model checker. Moreover, his approach focuses on formalizing design patterns, whereas our work deals with a more general approach based on design components [16] and their composition.

5 CONCLUSIONS AND FUTURE WORK

Discovering design errors is very important in that a small piece of design may be mapped to thousands of lines of implementation code. Finding these errors at the implementation stage might incur much higher expense and more resource, because these errors may hide in the complex implementation structures, thus are very difficult to detect.

We illustrate our analysis techniques through a case study on the analyses of the composition of hypermedia design components. However, our analyses are not restricted to the hypermedia domain. We have analyzed a design on general system sort [1]. We also developed a method to prove structural and behavioral correctness in the generic design composition [9].

Our approach showed that the structural and behavioral aspects of design components can be described in the same formal language so that we can analyze the properties of both aspects of design composition with the same method. This also opened the door to many interesting and important design analyses as, for instance, the interactions between the structural and behavioral aspects of software design as illustrated in our case study. Our approach has several advantages: first, it allows us to find errors in the design composition early in the development process and save expense to correct them later. Second, it provides mechanisms to achieve automated verification of the properties of software design. Third, it promotes reuse since the generic representations of design components can be stored in a repository and retrieved for instantiation and integration in an application. Fourth, as the composition of components can be treated as a component, the design analysis can be scaled up incrementally to large component-based software systems. This also addresses the state explosion problem in model checking by incrementally modeling components and checking their compositions. Fifth, the change of the design compositions can be achieved by simply modifying limited arguments in the descriptions of the integration of components as shown in Figure 9. The structural design evolutions can be achieved by applying corresponding Prolog rules.

Our analysis approach is limited to the kinds of properties that can be proved using Prolog and the highly expressive μ-calculus temporal logic. In principle, as a result of the experiments we have done so far, these underlying deductive facilities seem to be adequate for the purposes at hand. Besides verifying structural, behavioral and evolutionary properties, we are currently defining other classes of properties that we can use in our analysis of design compositions. These classes may include the properties about (real-)time, event ordering and access control. We are also working on techniques to map the XL counter-examples back onto the original structural and behavioral descriptions and assessing how much of this feature can be automated and how much effort is required to be able to show which parts of the structural and behavioral descriptions need to be revised under the light of the counter-examples.

The analysis results shown in this paper indicate that it could be highly beneficial to pursue further research on how the structural design changes may affect the behavior of software system composition, and how these changes can affect various properties of the design composition. These analyses cannot only provide guidance on important design decisions, but also be documented to provide information on the rationale behind these decisions.

References

[1] Paulo Alencar, Donald Cowan, Jing Dong, and Carlos Lucena. A Pattern-Based Approach to Structural Design Composition. *Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference (COMPSAC), Phoenix USA*, pages 160–165, October 1999.

[2] Paulo Alencar, Donald Cowan, Jing Dong, and Carlos Lucena. An Evolutionary Approach to Structural Design Composition. *Technical Report CS-99-16, Computer Science Department, University of Waterloo*, 1999.

[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] D. Batory and B.J. Geraci. Validating Component Composition in Software System Generators. *Proceedings of the 4th International Conference on Software Reuse*, pages 72–81, April 1996.

[5] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[7] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computer Surveys*, 28(4), December 1996.

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

[9] Jing Dong, Paulo Alencar, and Donald Cowan. Ensuring Structure and Behavior Correctness in Design Composition. *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems(ECBS), Edinburgh UK*, pages 279–287, April 2000.

[10] B. Fraser, J. Roberts, G. Pianosi, P. Alencar, D. Cowan, D. Germán, and L. Nova. Dynamic Views of SGML Tagged Documents. *Proceedings of the ACM SIGDOC*, pages 93–98, Sept. 1999.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.

[12] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts. *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, April 1995.

[13] Daniel M. Germán and Donald D. Cowan. Towards a Unified Catalog of Hypermedia Design Patterns. *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, Jan. 2000.

[14] Scott A. Hissam. Experience Report: Correcting System Failure in a COTS Information System. *Proceedings of the International Conference on Software Maintenance, Bethesda, USA*, pages 170–176, Nov. 1998.

[15] Daniel Jackson and Allison Waingold. Lightweight Extraction of Object Models from Bytecode. *Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA*, pages 194–202, May 1999.

[16] 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.

[17] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitalille, and Patrick Pagé. Pattern-Based Reverse-Engineering of Design Components. *Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA*, pages 226–235, May 1999.

[18] D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

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

[20] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[21] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient Model Checking Using Tabled Resolution. *Proceedings of the 9th International Conference on Computer Aided Verification (CAV), Haifa Israel, LNCS1243, Springer-Verlag*, July 1997.

[22] C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. XMC: A Logic-Programming-Based Verification Toolset. *Proceedings of the International Conference on Computer Aided Verification (CAV), LNCS1855, Springer-Verlag*, July 2000.

[23] Dirk Riehle. Composite Design Patterns. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA), USA*, pages 218–228, October 1997.

[24] Gustavo Rossi, Daniel Schwabe, and Alejandra Garrido. Design Reuse in Hypermedia Applications Development. *Proceedings of the ACM International Conference on Hypertext*, pages 57–66, April 1997.

[25] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison Wesley Longman, Reading, Mass., 1998.

[26] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 1996.

[27] XSB. The XSB Logic Programming System, Version 2.1. *Available from http://www.cs.sunysb.edu/~sbprolog*, 1999.