

# A behavioral analysis and verification approach to pattern-based design composition

Jing Dong<sup>1</sup>, Paulo S.C. Alencar<sup>2</sup>, Donald D. Cowan<sup>2</sup>

<sup>1</sup>University of Texas at Dallas, Department of Computer Science, Richardson, Texas 75083, USA;  
E-mail: jdong@utdallas.edu

<sup>2</sup>University of Waterloo, School of Computer Science, Waterloo, Ontario, Canada N2L 3G1;  
E-mail: {palencar,dcowan}@csg.uwaterloo.ca

Received: 14 September 2002/Accepted: 22 August 2003

Published online: 1 April 2004 – © Springer-Verlag 2004

**Abstract.** Integrating software components to produce large-scale software systems is an effective way to reuse experience and reduce cost. However, unexpected interactions among components when integrated into software systems are often the cause of failures. Discovering these composition errors early in the development process could lower the cost and effort in fixing them. This paper introduces a rigorous analysis approach to software design composition based on automated verification techniques. We show how to represent, instantiate and integrate design components, and how to find design composition errors using model checking techniques. We illustrate our approach with a Web-based hypermedia case study.

**Keywords:** Design patterns – Software design – Software components – Software specification – Verification – Model checking – Hypermedia systems

---

## 1 Introduction

Use of previously developed components in building software systems is an appealing idea because of the apparent reduction in cost and effort. Use of components should also lead to faster time-to-market for complex software applications. Further, since these components have probably been tested in use and may have even been formally validated they should produce a more robust software system. The task of integrating components at the design level requires the behavior, which is given in terms of the services and inter-connectivity between these components, to be composed without compromising system integrity and invariant. When the composition is inadequate to accomplish this, mostly because of unanticipated interactions among the components, software failures are

introduced and the system becomes unreliable. Our main goal is to provide techniques that allow component composition at the design level in order to reduce or prevent these integration problems. Previous research [11] indicates that deep knowledge about the domain and about the software design is a critical factor in the construction and integration of such applications.

Software components are often selected based on their guarantee of critical functional, fault-tolerant, real-time, and performance properties. Proving such properties still hold after the composition of these components can increase our confidence in 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. There is an increasing interest in modeling software by various formalisms and checking properties or finding errors against the models by model checkers [5]. Numerous examples can be found [6] in areas such as requirement analysis, distributed cache coherence analysis, word processor design analysis, mobile IP protocol analysis, CAD algorithm analysis, and Java meta-locking algorithm analysis.

At the architectural design level, object-oriented and Web-based frameworks use design components as development building blocks. Design components [15], such as object-oriented design patterns [13], have been proposed to reify good design practice from conceptual design building blocks into a tangible and composable form. Design components focus on component-based problem solving instead of component-based implementation. There has been substantial interest in discovering and documenting such reusable design experience in different domains as, for example, in hypertext design [22]. On the other hand, there has been little work on reasoning about the interactions among the design patterns when they have been composed.

In this paper, we introduce a rigorous analysis approach to software design composition based on automated verification techniques. The approach involves generic modeling formalism for design components and a framework in which the composition of such components can be verified. The main contributions can be summarized: (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; (ii) a process that supports the verification of design components in order to analyze through model checking whether behavioral composition properties hold; (iii) a case study applying the systematic approach to specify and verify the composition of design components in the Web-based hypermedia domain.

We have chosen to illustrate the verification framework through the design of a Web-based hypermedia documentation system. Hypermedia systems when used to deliver software product information are quite complex software systems themselves. Component-based approaches [15, 18] have been proposed as a promising solution to some of the problems that designers and integrators face when building their systems. According to these approaches, the hypermedia system is divided into components that address different hypermedia concerns such as content creation and presentation issues. The LivePage Web-based information system [10] is an example of a solution that follows this trend.

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 Web-based hypermedia design by checking whether relevant behavioral system properties hold. The last two sections are related work and conclusions.

## 2 Analysis technique

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

### 2.1 Overview

*Modeling techniques.* An overview of our model-based approach is shown in Fig. 1. Note that we separate the structural aspects and the behavioral aspects of a design. Initially, we model the design components using UML [4]. The structural and behavioral aspects of the design components are described through class diagrams and collaboration diagrams respectively. In addition, we also use class diagrams to represent the structural evolution of design components. Then, the structural and evolutionary aspects of each design component are transformed into Prolog representations and the behavioral aspects are transformed into process algebra representations. The representation of both structural and behavioral aspects of all design components in an application is then combined to form the design composition model. Properties are described in temporal logic and can be checked against the model. Since graphical notations are more intuitive and generally easier to understand, we use such notations as the method for designers and integrators to build their designs. In this way, they are not directly exposed to the intricacies of the underlying formalisms. On the other hand, the informal graphical models, which lack precision and may be ambiguous, cannot be the bases for a formal analysis. A formal model allows us to reason about the properties of design compositions and to verify our designs before system implementation.

*Analysis process.* Figure 2 illustrates the main characteristics of the analysis process underlying our approach.

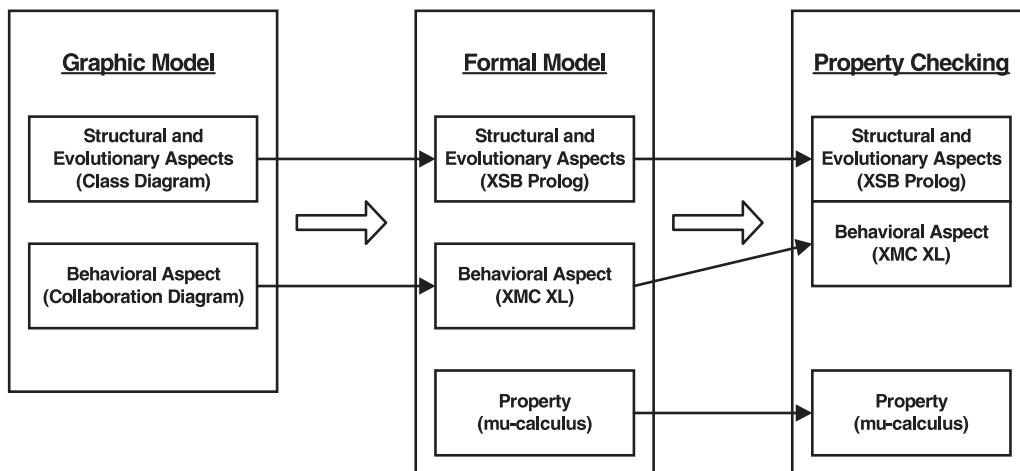


Fig. 1. Model overview

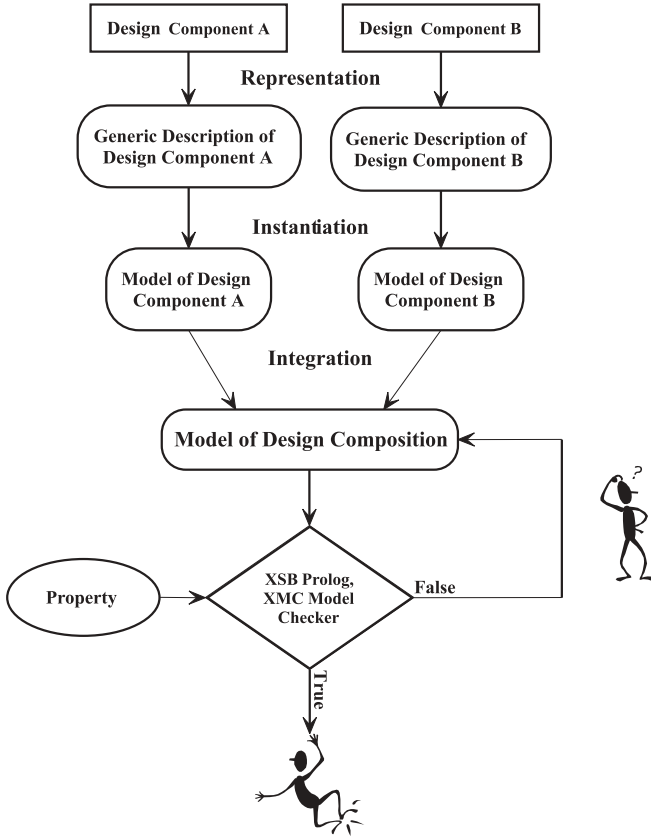


Fig. 2. The design analysis process

Design components are represented through declarations using Prolog and XL. The representations are generic in the sense that they capture good design practice in a domain-independent way. These declarations, 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 into a model of the design composition that is then subjected to automated verification. Formal automated verification techniques constitute the core of our analysis approach. Essentially, we rely upon Prolog proofs and model checking techniques [5] to analyze the composition and integration of design components. Model checking entails comparing two formal objects  $(\Sigma, \phi)$ , e.g. the software design components and their compositions are portrayed as a logic model,  $\Sigma$ , and the properties of these components are represented as logic formulas,  $\phi$ . One assumes that if a formula  $\phi$  is true in the model  $\Sigma$ , then the corresponding property holds in the model of the design. We use a model checker as a black box to check  $\Sigma$  against the property specification,  $\phi$ . The model checker outputs either true, if  $\Sigma$  satisfies  $\phi$ , or a counterexample, if it does not. When a property 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 a number of temporal logics.

XMC [20] is a model checker for verifying temporal properties of a system. It is written in the XSB table Prolog programming system [26]. Temporal properties are expressed in the alternation-free fragment of the  $\mu$ -calculus [14], a very expressive temporal logic; 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 [17, 23]. Prolog terms and predicates are used respectively to represent values and computations. Thus, specifications can make use of recursive data structures and computations. When a XL specification of a system is loaded in XMC, the user can write properties related to the system in  $\mu$ -calculus and run XMC to check these properties against the XL specification automatically. If the properties hold, XMC will reply “yes”, otherwise “no”. XMC has been successfully used to verify various software systems as documented in [21].

In the next subsections, we describe each phase of our design analysis approach. In particular, we show how to represent, instantiate and integrate design components. We also illustrate how to verify composition properties of these components through a case study. Since we have described the representation and verification of the structural and evolutionary aspects of the design components in [1], we focus our attention on the behavioral aspect and verifying behavioral composition properties expressed in the  $\mu$ -calculus temporal logic using the XMC model checker in this paper.

## 2.2 Representation

Initially, design components, such as design patterns, are represented in UML diagrams. The design component’s behavior is represented using UML collaboration diagram. The behavioral aspects of the design components are then transformed into XL, the model specification language for XMC.

The syntax of the XL specification is given next:

```

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. Operation ‘o’ is sequential composi-

tion; ‘#’ is nondeterministic choice; ‘|’ is parallel composition; ‘@’ is relabeling where “PortMap” is a list of substitutions; and ‘\’ is restriction where “PortList” is a list of port names. 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. “Comp” is a computation, e.g.,  $X$  is  $Y+1$ . Process “if (Comp, Pexp, Pexp)” behaves like the first “Pexp” if computation “Comp” succeeds and otherwise like the second “Pexp”. Recursion is provided by a set of process definitions, “Pdef”, of the form “Pname ::= Pexp.”

The behavior of a design component is defined in terms of processes and their communications in Definitions 1 and 2. An object is modeled as a process. The interactions among objects are modeled as process communications. A process may have input and output ports which can send (or receive) input (or output) messages. A process can also perform some actions.

**Definition 1.** *The behavioral aspect of a design component BDC is a tuple  $BDC = \langle O, P, IP, OP, IM, OM, IM_I, OM_I, A \rangle$ , where*

- $O$  is a finite set of object names. These objects are the instances of classes in the design component.
- $P$  is a finite set of process names. The behavior of each object  $o \in O$  is modeled as a process  $p \in P$  in XL, where  $P : O \leftrightarrow P$ . We denote the process for object  $o \in O$  by  $P(o)$ .
- $IP$  is a finite set of input ports attached to a process. A process can input messages from its input ports, where  $iport : IP \rightarrow P$ . We denote the set of input ports for the process  $p \in P$   $IP(p) = \{i \in IP \mid iport(i) = p\}$ .
- $OP$  is a finite set of output ports attached to a process. A process can output messages from its output ports, where  $oport : OP \rightarrow P$ . We denote the set of output ports for the process  $p \in P$  by  $OP(p) = \{o \in OP \mid oport(o) = p\}$ .
- $IM$  is a finite set of input messages sent to a process, and  $imessage : IM \rightarrow IP$ . We denote the set of input message sent to the process  $p \in P$  from other processes by  $IM(p)$ . These input messages are received from one or more input ports of the process  $p$ . We denote the set of messages received via input port  $ip \in IP$  of process  $p \in P$  by  $IM(ip, p) = \{i \in IM \mid ip \in IP(p), imessage(i) = ip\}$ .
- $OM$  is a finite set of output messages sent from a process, and  $omessage : OM \rightarrow OP$ . We denote the set of output message sent from the process  $p \in P$  to other processes by  $OM(p)$ . These output messages are sent out from one or more output ports of the process  $p$ . We denote the set of messages sent via output port  $op \in OP$  of process  $p \in P$  by  $OM(op, p) = \{o \in OM \mid op \in OP(p), omessage(o) = op\}$ .

- $IM_I$  is the finite set of input messages sent from outside the design component to a process  $p \in P$ , denoted by  $IM_I(p)$  and  $IM_I(p) \subset IM(p)$ .
- $OM_I$  is the finite set of output messages sent outside the design component from a process  $p \in P$ , denoted by  $OM_I(p)$  and  $OM_I(p) \subset OM(p)$ .
- $A$  is a set of finite actions that can be performed by a process. We denote the set of actions of the process  $p \in P$  by  $A(p)$ . An action is used to model a method of an object.

**Definition 2 (XL-Process).** *Let  $BDC = \langle O, P, IP, OP, IM, OM, IM_I, OM_I, A \rangle$  be the behavior of a design component, and assume the following auxiliary definitions:  $A(p, i)$  is the set of actions that are performed when the process  $p$  receives a message  $i$ , where  $IP(i)$  is the input port of message  $i$ ;  $OM(p, i)$  is the set of messages that are sent out by process  $p$  when it receives a message  $i$ , where  $OP(OM(p, i))$  is the output port of message  $OM(p, i)$ ;  $P(p, i)$  is the set of processes that are executed by process  $p$  when it receives a message  $i$ . Then, the XL-Process  $XL(BDC)$  induced by the behavior of a design component BDC is defined by introducing, for each process  $p \in P$  an equation:*

$$BDC(p) = \#_{\forall i \in IM(p)} (in(IP(i), i) \circ action(A(p, i)) \circ P(p, i) \circ out(OP(OM(p, i)), OM(p, i))).$$

Thus,

$$XL(BDC) = \big|_{\forall p \in P} BDC(p).$$

Informally speaking, the behavior of a design component,  $XL(BDC)$ , is defined as the parallel composition of XL process statements,  $BDC(p)$ , where  $p$  is a process. Each XL statement is a non-deterministic choice of a list of sequential compositions of input, action, process and output.

### 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 pattern component with terms representing domain information. Meanwhile, the integration of two design components can be achieved by overlapping their common parts. There are some issues to be considered during the integration phase. First, some classes/objects may play different roles when they are the shared parts of two different design patterns. We have to make sure that the generic elements in these shared parts are instantiated with the same names in the corresponding XL rules. Second, the integration of two or more design patterns may cause undesired interactions among them. Some properties or constraints of a design pattern may not hold after integration.

Component instantiation can be based on theory interpretation, a formal approach to refinement. Composition can also be based on theory interpretation and on criteria to ensure that the composition of design components is correct. However, in this paper, instead of focusing on criteria for instantiation and composition, we concentrate on the verification of composition properties. A more detailed description of the composition techniques can be found in [7].

### 2.4 Property checking

As we have previously mentioned, we will verify behavioral composition properties expressed in the  $\mu$ -calculus temporal logic using the XMC model checker.

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

$$\begin{aligned}
 D &\rightarrow Z+ = F \text{ (least fixed point)} \\
 &\quad | Z- = F \text{ (greatest fixed point)} \\
 F &\rightarrow Z|tt|ff|F \vee F|F \wedge F|\langle A \rangle F|[A]F
 \end{aligned}$$

“ $D$ ” is a  $\mu$ -calculus formula, which is defined in the format of either “ $Z+ = F$ ” or “ $Z- = F$ ”. “ $Z$ ” is a set of formula variables encoded as Prolog atoms; “ $tt$ ” and “ $ff$ ” are propositional constants;  $\vee$  and  $\wedge$  are standard logical connectives; “ $A$ ” is a set of actions; “ $\langle A \rangle F$ ” denotes that possibly after the action “ $A$ ” the formula “ $F$ ” holds; “ $[A]F$ ” denotes that necessarily after the action “ $A$ ” the formula “ $F$ ” holds. For example, the deadlock property can be described in XMC as follows: “ $deadlock+ = [-]ff \vee \langle - \rangle deadlock$ ” where a system is in deadlock state  $X$  if the system cannot progress ( $ff$ ) after any action ( $[-]$ ), or it is eventually possible on some paths ( $\langle - \rangle$ ) deadlock state  $X$  is reachable.

### 3 Case study

In this section, we first describe two hypermedia design components, then analyze their compositions by repre-

sentation, instantiation and integration of these components. Properties are checked against the behavioral aspect of the composition model. We also check whether behavioral properties hold when a design component evolves by the addition of a component element. We have adopted the case study related to the design of the LivePage Web-based information system [10] for hyper-text document management.

#### 3.1 Design components

Hypermedia design patterns [22] 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 comprehensive catalog of hypermedia design patterns can be found in [12]. In the following, we will use the Active Reference pattern and the Navigational Contexts pattern, as examples, to show the description of these hypermedia design pattern components and their composition, and to verify the properties by a model checker (XMC).

The Active Reference pattern [22] is used to help the user to have visual knowledge about the current location in terms of spatial or time space during the navigation, and allow the user to change to other positions in the complex navigation space. The UML class diagram of this pattern is shown in Fig. 3. 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 such as 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

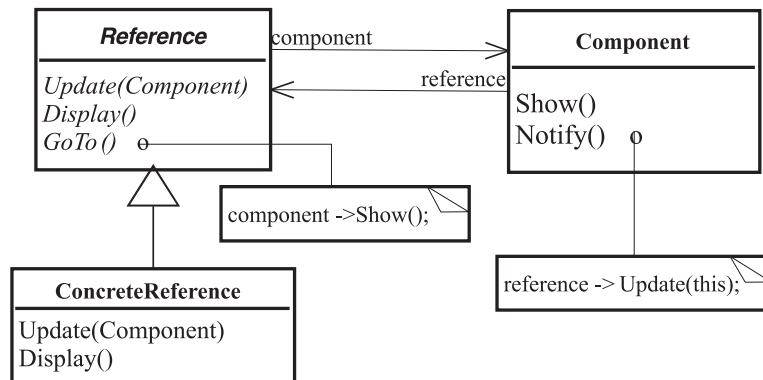


Fig. 3. Active reference pattern (class diagram)

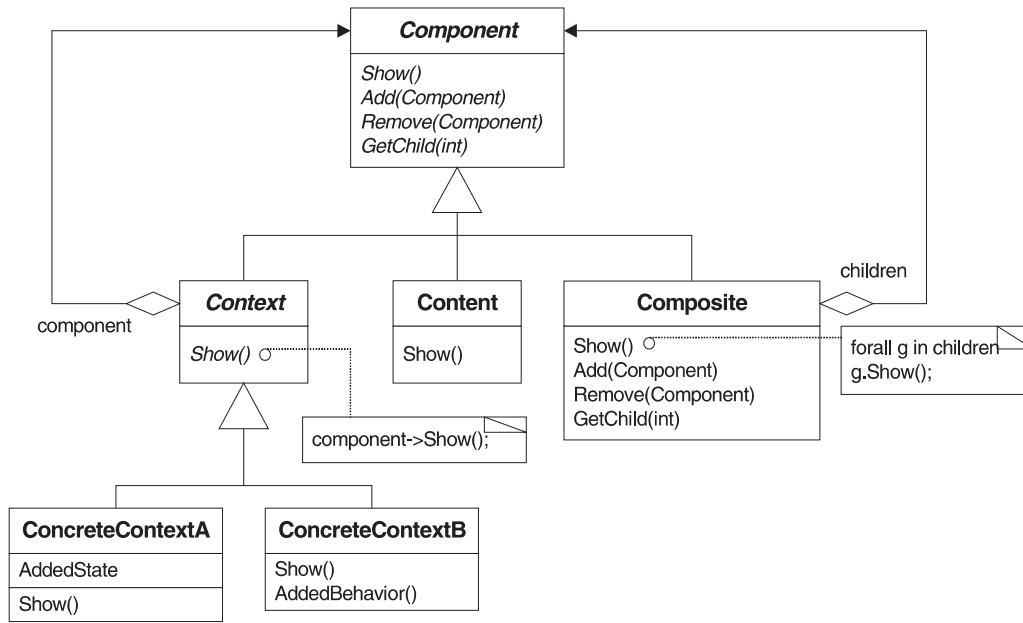


Fig. 4. Navigational contexts pattern (class diagram)

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 Navigational Contexts pattern [22] 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 in a manner similar to the Decorator pattern [13]. If the collections of hypermedia components are modeled as an aggregate similar to that in the Composite pattern [13], the Navigational Contexts pattern can be seen as the integration of the Decorator pattern and the Composite pattern, which is shown in Fig. 4. The names of some classes and operations have been changed to represent the corresponding meanings in the Navigational Contexts pattern. For example, 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

In this section, we describe the behavioral aspects of the Active Reference and the Navigational Contexts patterns in XL. These formal representations are transformed from the UML collaboration diagrams. Note that the representations of the structural and evolutionary aspects of these patterns can be found in [8].

The UML collaboration diagram shown in Fig. 5 describes the behavioral aspect of the Active Reference pattern, which contains four messages between two objects. The vertices of this diagram describe the objects that participate in the collaboration in this pattern. The arcs of this graph represent the links that connect these objects. The messages sent among these objects where each message shown as a line with an arrowhead in Fig. 5 represents a communication between two objects. The messages “A1:r.GoTo” and “A2:c.Show” present a first sequence of operations where “A1” and “A2” define the time order of these two operations. The messages “B1:c.Notify” and “B2:r.Update” present a second sequence of operations in a similar way.

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 based on Definitions 1 and 2 in Sect. 2.2. The XL processes are 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)).
```

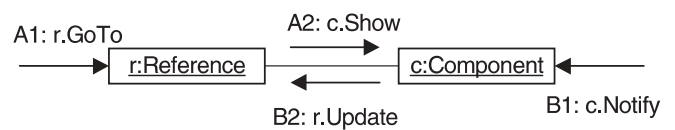


Fig. 5. Active reference pattern (collaboration diagram)

```
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)
  o cShow(Reference, GoTo, Component, Show, C) }
  |{ cNotify(Component,Notify,Reference,Update, R)
  o rUpdate(Component,Notify,Reference,Update, R)}.
```

Each XL process describes the behavior of an object. 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.GoTo` in Fig. 5) defines a process that sends the message “(Reference, GoTo, Component, Show)” to a channel “C” and performs an action. The “ref” process defines the behavior of this pattern as the parallel composition of these processes.

Similarly, the behavioral aspect of the Navigational Contexts pattern is modeled in terms of the collaboration (see Fig. 6) among the methods in the classes of this pattern in XL based on Definition 1 and 2:

```
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{ xShow(Context, Component, Concrete, Show)
  # nShow(Content, Show)
```

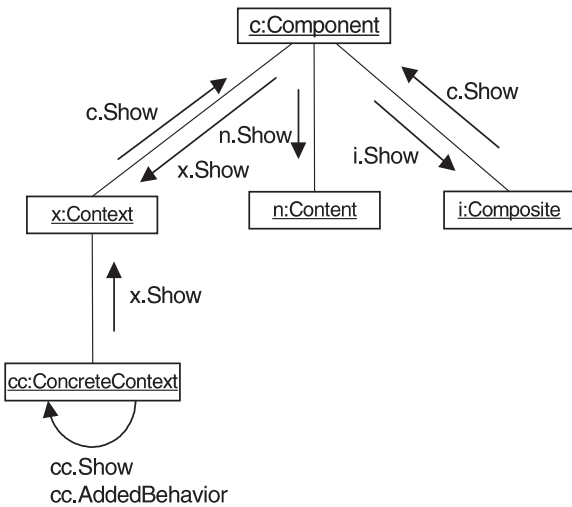


Fig. 6. Navigational contexts pattern (collaboration diagram)

```
# iShow(Composite, Component, Show))
o cShow(Context, Component, Concrete, Content,
  Composite, Show, P, T).
```

```
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 (see Fig. 7). The behavioral aspect can be instantiated as following: “ref(reference, painting, goTo, update, show, notify)”.

As another example, the Navigational Contexts pattern can be instantiated for a user to explore the paintings in a museum in different contexts through context links. The behavioral aspect can be instantiated as: “nav(context, [button], painting, content, composite, show)”. Therefore, the design decision and information of the Active Reference pattern and the Navigational Contexts pattern are written in the XSB Prolog database, which can be composed with those of other component instances.

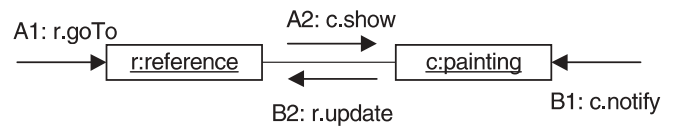


Fig. 7. An instance of the active reference pattern

### 3.4 Integration

Integration is the assembly of design components into a software system. In our approach, the representation of each design component is assembled within a XSB Prolog database to be the model of design composition.

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 Fig. 8, the “content” class is an overlapping part of the two design components.

The integrated design can be described as a process in XL as follows:

```
reference_context ::=
  ref(reference, content, goTo, update, show, notify)
  | nav(context, [button], painting, content, composite,
        show).
```

Essentially, this design, which we call “reference\_context”, is the parallel composition of the behavioral models of the two design components.

### 3.5 Property checking and design analysis

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

Behavioral properties include properties such as safety and liveness. For example, the system is deadlock free

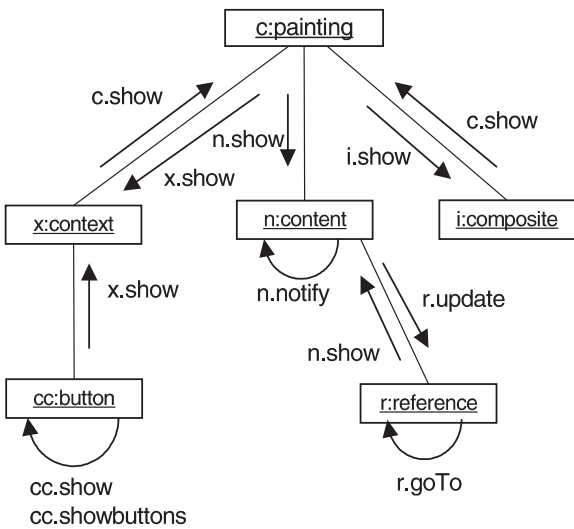


Fig. 8. The design composition (collaboration diagram)

(safety); an action is eventually performed when a button is pressed (liveness). In this section, we show examples of liveness properties.

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 “reference” class. Therefore, the invocation of the “goTo” operation should eventually invoke both the “show” operation in the “content” class (“liveness1”) and the “show” operation in the concrete context class (“liveness2”), that is the “button” class. These two “show” operations display the content and the context information of a hypermedia component respectively. These two liveness properties are described generically in XMC as follows:

```
% Content::Show will be eventually invoked
% when Reference::GoTo is invoked.
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
  v form1(Content, Show)
  v [-] formula1(Content, Show).
```

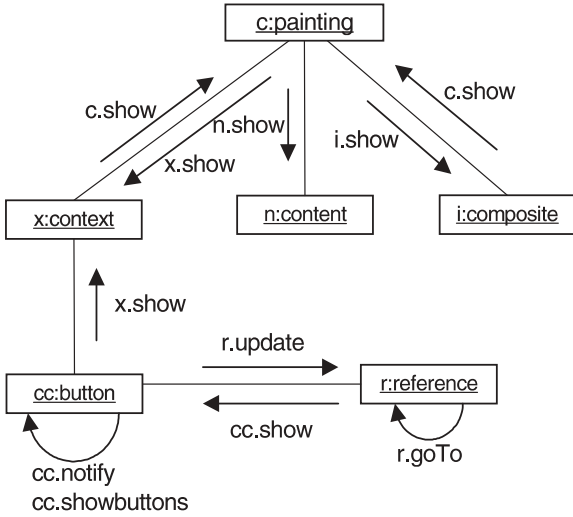
```
form1(Content, Show) +=
  <n_Show(Content, Show)> tt
  v [-{r_GoTo(,-)}] form1(Content, Show).
```

```
% Context::Show will be eventually invoked
% when Reference::GoTo is invoked.
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
  v form2(ConcreteContext, Show)
  v [-] formula2(ConcreteContext, Show).
```

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

These descriptions are instantiated to represent the liveness properties in this application as follows: “liveness1(reference, goTo, content, show)”. “liveness2(reference, 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 such as the map of a museum to change the current



**Fig. 9.** The modified design composition (collaboration diagram)

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. One solution to this problem is to move the overlapping part further down to the concrete context class (button) as shown in Fig. 9. This change can be easily achieved in our design composition model by updating the underlined part from “content” to “button” in the integrated design in XL described in Sect. 3.4. 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 in [8]. In the following, we will show one possible evolution of the previous design,

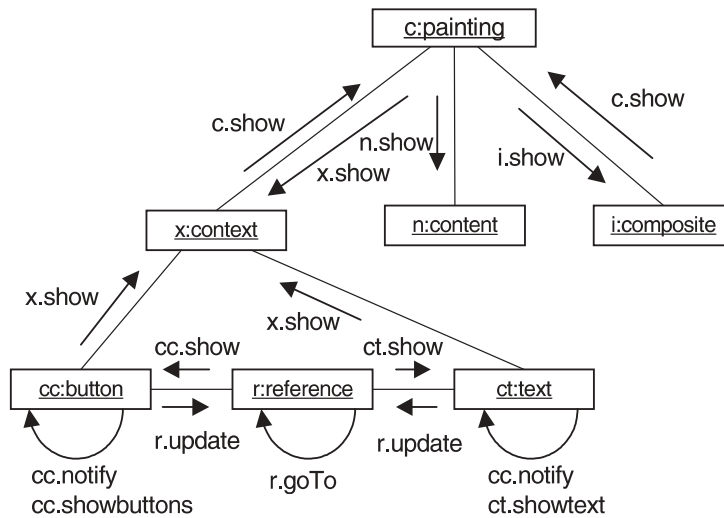
and check the behavioral property against this evolved design.

When the application requires another kind of context information, such as “text” information, in addition to “button” information, we can achieve this design decision of structural change by instantiating a predefined Prolog rule about the evolution [8]. The modified design is shown in Fig. 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 “reference” class is called. This property can be described by instantiating the second liveness property as: “liveness2(reference, 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 ensure other properties we have checked still hold after the design changes. For this case, we have also verified the “liveness1” property in the modified design and there was no reported error this time.

Our approach could also be used as a base to perform other kinds of analyses. As an example, accessibility to hypermedia content may be represented and verified using the approach based on logics. This feature can be helpful, for example, in the identification of content that has been defined but is not being used by the application (“dead content”). Furthermore, hypermedia Web sites may be checked for deadlocks because a deadlock can be exploited by hackers to generate a denial of service (DOS) attack [25].

#### 4 Related work

Riehle [19] proposed an analysis method for the composite design patterns. Role diagrams were introduced



**Fig. 10.** The evolved design composition (collaboration diagram)

to describe the patterns, and a role relation matrix was used to depict the composition constraints visually. This work was restricted to patterns based on object collaborations, and lacked generality and formal treatment of composition.

Formalizing design patterns and architecture patterns has been proposed in [2, 16]. Although Mikkonen [16] has discussed the composition of two design patterns based on a formal method, his approach relies on a specific specification language (DisCo). Correctness depends on the refinement correctness of this language since the composition is achieved in terms of refinement. Our approach emphasizes specifying design components and their compositions, and checking the properties by a model checker.

Taibi and Ngo [24] proposed a formal approach to the specification of design pattern composition, which is very similar to our approach in [7] in the specification aspect. Although formal specification of design patterns and their combinations allows rigorous reasoning about them, Taibi and Ngo did not investigate the verification of pattern combinations. In addition to formal specification of pattern combinations, we have investigated the manual verification techniques, such as theorem proving, in [7]. In this paper, we concentrate on automated verification techniques, e.g., model checking.

In [3], domain-independent algorithms are provided to validate component compositions for the GenVoca model of software generators. In addition to syntactic checking, they provided design rule (domain-specific constraint) checking to ensure semantic 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.

Other work on tool support for design patterns [9] also discussed constraints on patterns. Nevertheless, they worked on single pattern constraints at implementation level. Our work focuses the interactions among different patterns when they are integrated.

## 5 Conclusions and future work

In this paper we have introduced a rigorous analysis approach to software design composition based on automated verification techniques. We illustrate our analysis techniques through a case study on the composition of hypermedia design components. Our approach has several advantages. First, it allows us to find errors in the design composition early in the development process and save the costs of having to correct them later. Second, it provides mechanisms to achieve automated verification of the properties of software designs. Third, it promotes reuse since the generic representations of design components can be stored in a repository and retrieved for instantiation and integration in a specific application.

Our analysis approach is limited to the kinds of properties that can be proved using the highly expressive  $\mu$ -calculus temporal logic. In principle, as a result of the experiments we have done so far, these underlying deductive facilities seem to be adequate. In addition, the transformations from UML diagrams to formal representations are conducted manually. We are currently working on automated transformation tools. Besides verifying structural, behavioral and evolutionary properties, we are currently defining classes of properties that we can use in our analysis of design compositions. These classes may include properties about (real-)time, event ordering and access control.

## References

- Alencar P, Cowan D, Dong J, Lucena C (1999) A Pattern-Based Approach to Structural Design Composition. Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference (COMPSAC), Phoenix USA, pp 160–165, October
- Alencar PSC, Cowan DD, Lucena CJP (1996) A Formal Approach to Architectural Design Patterns. Proceedings of the Third International Symposium of Formal Methods Europe, pp 576–594
- Batory D, Geraci BJ (1996) Validating Component Composition in Software System Generators. Proceedings of the 4th International Conference on Software Reuse, pp 72–81, April
- Booch G, Rumbaugh J, Jacobson I (1999) The Unified Modeling Language User Guide. Addison-Wesley
- Clarke EM, Emerson EA, Sistla AP (1986) Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Transactions on Programming Languages and Systems 8(2):244–263, April
- Clarke EM, Wing JM (1996) Formal Methods: State of the Art and Future Directions. ACM Computing Surveys 28(4), December
- Dong J, Alencar P, Cowan D (2000) 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, pp 279–287, April
- Dong J (2002) Design Component Contracts: Model and Analysis of Pattern-Based Composition. Ph.D. Thesis, Computer Science Department, University of Waterloo, June
- Florijn G, Meijers M, van Winsen P (1997) Tool Support for Object-Oriented Patterns. Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), pp 472–495, June
- Fraser B, Roberts J, Pianosi G, Alencar P, Cowan D, German D, Nova L (1999) Dynamic Views of SGML Tagged Documents. Proceedings of the ACM SIGDOC, pp 93–98, September
- Garlan D, Allen R, Ockerbloom J (1995) Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts. Proceedings of the 17th International Conference on Software Engineering, pp 179–185, April
- German DM, Cowan DD (2000) Towards a Unified Catalog of Hypermedia Design Patterns. Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, January
- Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company
- Kozen D (1983) Results on the Propositional  $\mu$ -calculus. Theoretical Computer Science, 27:333–354
- Keller RK, Schauer R (1998) Design Components: Towards Software Composition at the Design Level. Proceedings of the 20th International Conference on Software Engineering, pp 302–311

16. Mikkonen T (1998) Formalizing Design Pattern. Proceedings of the 20th International Conference on Software Engineering, pp 115–124
17. Milner R (1989) Communication and Concurrency. International Series in Computer Science. Prentice Hall
18. Nierstrasz O, Dami L (1995) Component-Oriented Software Technology. In: Nierstrasz O, Tschritzis D (eds) Object-Oriented Software Composition. Prentice Hall, pp 3–28
19. Riehle D (1997) Composite Design Patterns. Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA), USA, pp 218–228, October
20. Ramakrishna YS, Ramakrishnan CR, Ramakrishnan IV, Smolka SA, Swift T, Warren DS (1997) Efficient Model Checking Using Tabled Resolution. Proceedings of the 9th International Conference on ComputerAided Verification (CAV), Haifa Israel. LNCS, vol 1243. Springer-Verlag, pp 143–154, July
21. Ramakrishnan CR, Ramakrishnan IV, Smolka SA (2000) XMC: A Logic-Programming-Based Verification Toolset. Proceedings of the International Conference on Computer Aided Verification (CAV). LNCS, vol 1855. Springer-Verlag, pp 576–580, July
22. Rossi G, Schwabe D, Garrido A (1997) Design Reuse in Hypermedia Applications Development. Proceedings of the ACM International Conference on Hypertext, pp 57–66, April
23. Stirling C (1991) An Introduction to Modal and Temporal Logics for CCS. Lecture Notes in Computer Science, vol 491. Springer-Verlag, pp 1–20
24. Taibi T, Ngo D (2003) Formal Specification of Design Pattern Combination Using BPSL. Information and Software Technology 45, Elsevier, pp 157–170
25. Wang W, Hidvegi Z, Bailey AD, Whinston AB (2000) E-Process Design and Assurance Using Model Checking. IEEE Computer 33(10):48–53, October
26. XSB (1999) The XSB Logic Programming System, Version 2.1. Available from <http://www.cs.sunysb.edu/~sbprolog>