A Behavioral Analysis Approach to Pattern-Based 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

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 certainly 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 since hypermedia documents and systems are now becoming complex software applications, which are component-based.

Keywords: Software design and analysis, hypermedia design patterns, component-based software engineering, Web-based information systems, model checking, design components.

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 [13] 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 [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, 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 [12], 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

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

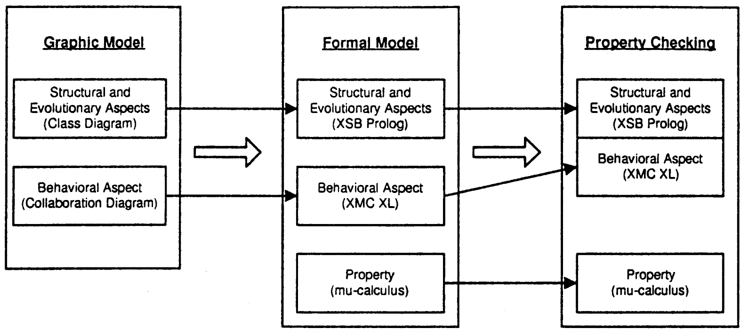
Graphic Model
Structural and Evolutionary Aspects (Class Diagram)
Behavioral Aspect (Collaboration Diagram)
Formal Model
Structural and Evolutionary Aspects (XSB Prolog)
Behavioral Aspect (XMC XL)
Property (mu-calculus)
Property Checking
Structural and Evolutionary Aspects (XSB Prolog)
Behavioral Aspect (XMC XL)
Property (mu-calculus)

Figure 1: Model Overview

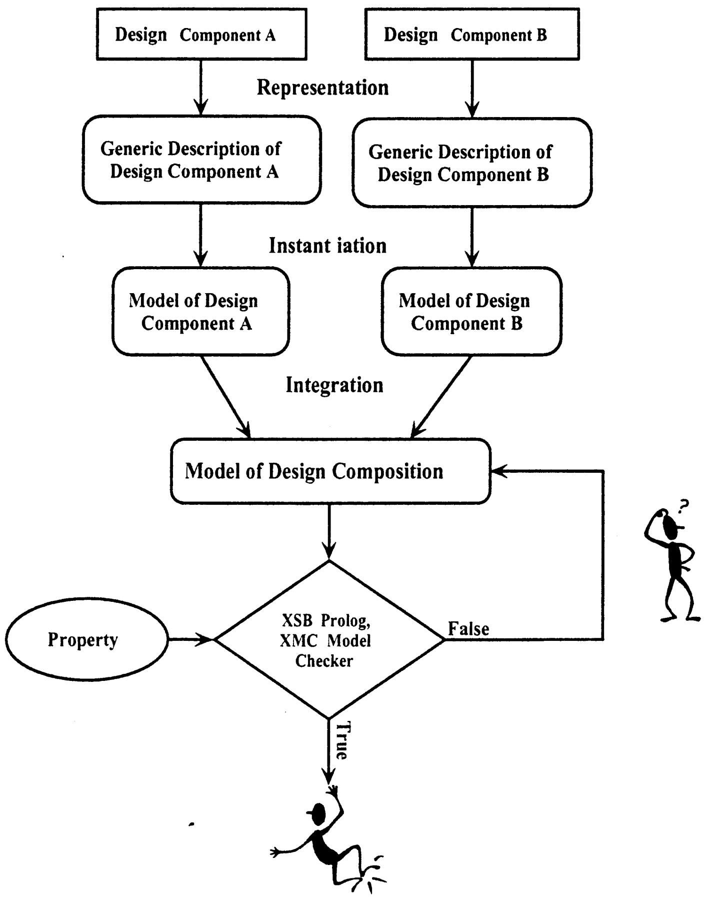
Design Component A
Design Component B
Representation
Generic Description of Design Component A
Generic Description of Design Component B
Instant iation
Model of Design Component A
Model of Design Component B
Integration
Model of Design Composition
Property
XSB Prolog, XMC Model Checker
False
True
?

Figure 2: The Design Analysis Process

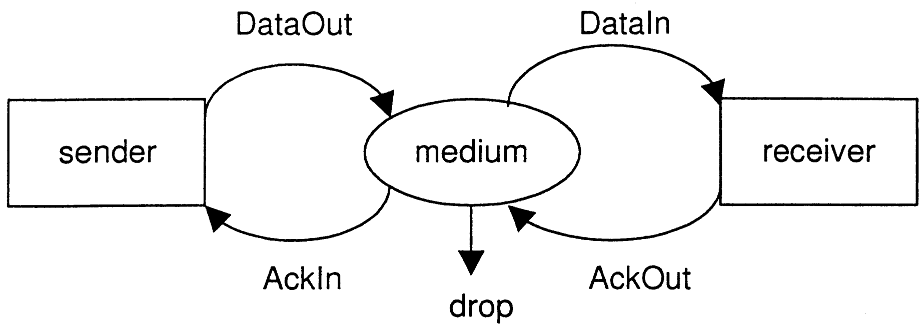

Figure 3: Alternating Bit Protocol

and send a new message. However, the *sender* may be timeout while waiting for the acknowledgement with a certain sequence number, and resends the current message. The specification of the Alternating Bit Protocol in XL is shown as following. 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;
                Action(sent);
                sender(AckIn, DataOut, NSeq)
            }
            %% unexpected ack, resend message
            else sender(AckIn, DataOut, Seq)
    #
        %% upon timeout, resend message
        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);
```

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

3 Case Study

In this section, we first describe two hypermedia design components, then analyze their compositions by representation, 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 [11] for hypertext document management.

3.1 Two Design Components

Hypermedia design patterns [21] 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 [14]. 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 [21] is used to help an 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 Figure 4. 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 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.

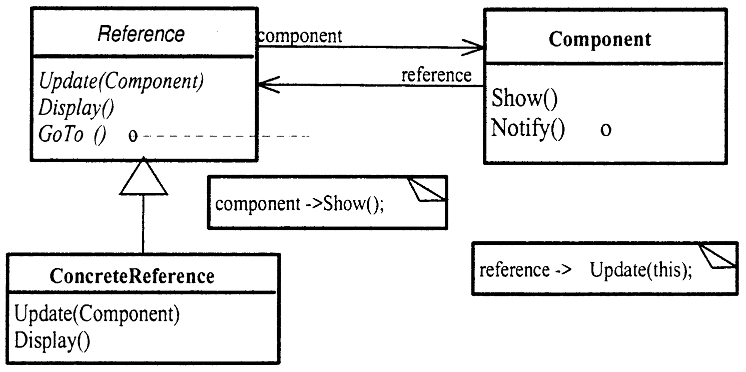

Figure 4: Active Reference Pattern (Class Diagram)

The Navigational Contexts pattern [21] 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 [12]. If the collections of hypermedia components are modeled as an aggregate similar to that in the Composite pattern [12], the Navigational Contexts pattern can be seen as the integration of the Decorator pattern and the Composite pattern, which is shown in Figure 5. The

If the message is equal to `Update`, the process performs an action. The `component` process is defined similarly. The `actRefBehavior` process defines the behavior of this pattern as the parallel composition of these processes.

```
reference(In1, In2, Out, GoTo, Show, Update, R) ::=
  in(In1, Op);
  if Op == GoTo
  then {
    action(rGoTo(R, GoTo));
    out(Out, Show)
  }
  in(In2, Op);
  if Op == Update
  then action(rUpdate(R, Update)).

component(In1, In2, Out, Show, Notify, Update, C) ::=
  in(In1, Op);
  if Op == Show
  then action(cShow(C, Show));
  in(In2, Op);
  if Op == Notify
  then {
    action(cNotify(C, Notify));
    out(Out, Update);
  }

actRefBehavior(GoTo, Show, Notify, Update, R, C) ::=
    reference(C2R, Clnt1, R2C, GoTo, Show, Update, R)
  | component(R2C, Clnt2, C2R, Show, Notify, Update, C).
```

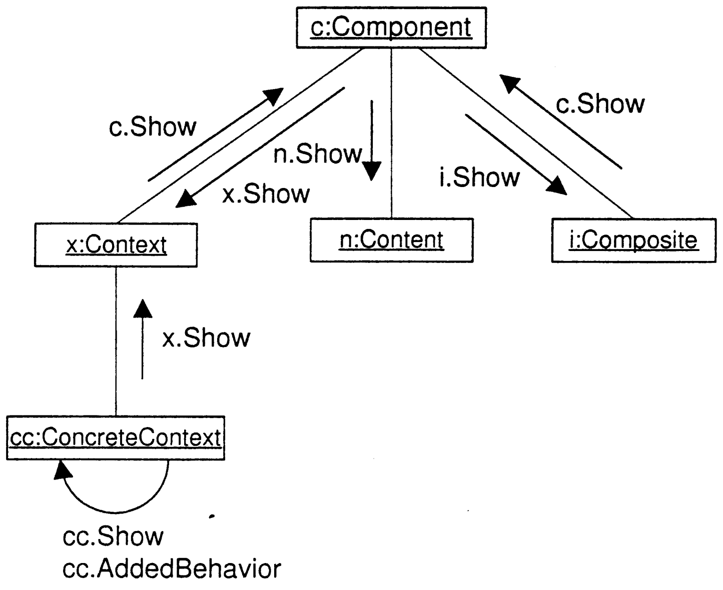

Figure 7: Navigational Contexts Pattern (Collaboration Diagram)

Similarly, the behavioral aspect of the Navigational Contexts pattern is modeled in terms of the collaboration (see Figure

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 Figure 8). The behavioral aspect can be instantiated as following: `actRefBehavior(goTo, show, notify, update, reference, painting)`.

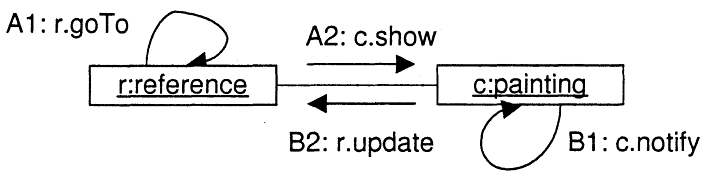

Figure 8: An Instance of the Active Reference Pattern

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: `navConBehavior(show, showbuttons, painting, composite, context, [button], content)`. 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.

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.

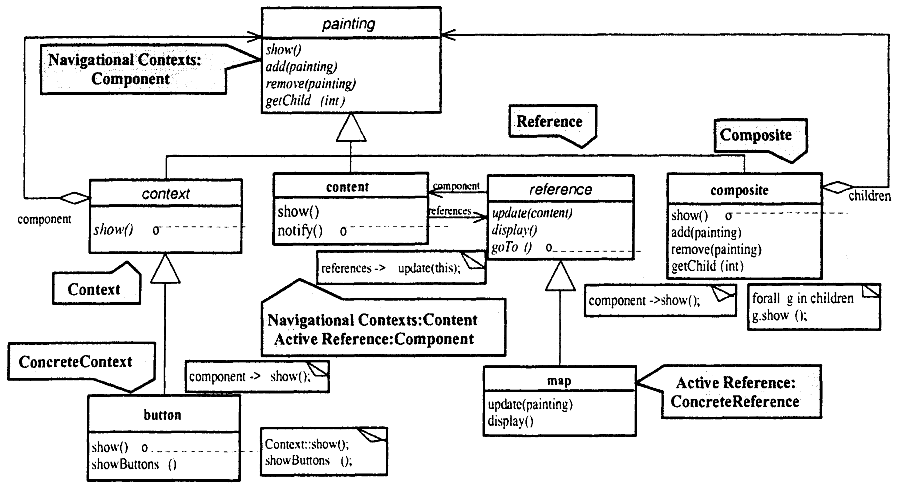

Figure 9: The Design Composition (Class Diagram)

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 9 (class diagram) and Figure 10 (collaboration diagram), the *content* class is an overlapping part 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) -=
      [rGoTo(Reference, GoTo)] formula1(Content, Show)
  /\ [-] liveness1(Reference, GoTo, Content, Show).

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

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

% Context::Show will be eventually invoked
% when Reference::GoTo is invoked.
liveness2(Reference, GoTo, ConcreteContext, Show) -=
     [rGoTo(Reference, GoTo)]
     formula2(ConcreteContext, Show)
  /\ [-] liveness2(Reference, GoTo, ConcreteContext, Show).

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

form2(ConcreteContext, Show) +=
      <ccShow(ConcreteContext, Show)> tt
  \/ [-{rGoTo(_,_)}] 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 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) as shown in Figure 12 (collaboration diagram). This change can be easily achieved in our design composition model by updating the underlined part from `content` to `button` in Figure 11. 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 [9]. 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 [9]. The modified design is shown in Figure 13. For brevity, we only show the collaboration diagram and omit the class diagram here. It contains one additional concrete context class (*text*). Therefore, we need to ensure the *show* of service (DOS) attack.

4 Related Work

Riehle [20] proposed an analysis method for the composite design patterns. Role diagrams were introduced 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 [3, 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.

In [4], 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 [10] 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.

UML collaboration diagrams have been used, as a basis, to define test criteria for static and dynamic testing at the design level in [1]. In contrast, we use collaboration diagrams as a basis for automated verification of design compositions.

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. Fourth, as the composition of components can be treated as a component, the design analysis can scale up incrementally to large component-based software systems. For this reason, the state explosion problem in model checking can be addressed by incrementally modeling components and checking their compositions.

Our analysis approach is limited to the kinds of properties that can be proved using 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. 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

[1] Aynur Abdurazik and Jeff Offutt. Using UML Collaboration Diagrams for Static Checking and Test Generation. *Proceedings of the Third International Conference on the Unified Modeling Language (UML), LNCS1939, Springer-Verlag*, pages 383–395, October 2000.

[2] 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.

[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.

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

[23] John Vlissides. Notation, Notation, Notation. *C++ Report*, April 1998.