

On Analysis of Design Component Contracts: A Case Study

Jing Dong
Department of Computer Science
University of Texas at Dallas
Richardson, Texas 75083, USA
jdong@utdallas.edu

Paulo Alencar and Donald Cowan
School of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3G1, Canada
{palencar,dcowan}@csg.uwaterloo.ca

Abstract

Software patterns are a new design paradigm used to solve problems that arise when developing software within a particular context. Patterns capture the static and dynamic structure and collaboration among the components in a software design. A key promise of the pattern-based approach is that it may greatly simplify the construction of software systems out of building blocks and thus reuse experience and reduce cost. However, it also introduces significant problems in ensuring the integrity and reliability of these composed systems because of their complex software topologies, interactions, and transactions. There is a need to capture these features as a contract through a formal model that allows us to analyze pattern-based designs. In this paper, we provide a formal framework for ensuring the integrity of the compositions in object-oriented designs by providing mathematically rigorous modeling and analysis techniques for object-oriented systems comprising pattern-based designs as the basic building blocks or design components. A case study related to a hypermedia web-based application is presented to illustrate our approach in distributed systems.

Keywords: Component-based software design, Process Calculus, Contract, Design Pattern, Model Checking, Web-based systems.

1 Introduction

Component-based software development focuses on building large software systems by integrating existing software components [20, 16]. At the foundation of this approach is the assumption that certain parts of large software systems reappear with sufficient regularity that common parts should be written once, and systems should be assembled through reuse rather than rewritten [16]. For successful component-based software development several prerequisites are needed, including a technology base in

form of component models, a selection of commercial of the shelf components, integration techniques, run-time environments, development methods, and development tools.

Design patterns [14] capture the distilled experience of expert designers. The expectation is that expert designers will have used similar proven designs to resolve similar problems in different application domains. Patterns document these proven designs, removing domain-specific features thus specifying only their essential aspects. A documented pattern, then, is deployable in a new domain via the addition of domain-specific features to the pattern's essential features. Since a design pattern is a recurring piece of software design, it can be seen as a component, called a design component in [20], and used 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.

To build software systems, a designer needs to solve many problems. Applying known design patterns to address these problems allows the designer to take advantage of expert design experience documented in each pattern. Thus, software system design becomes the composition of many design patterns, as shown in [20, 9, 10, 7]. However, simply putting patterns together may result in failures because they can have undesired interactions. Failures to detect these interactions may result in incorrect design decisions. If design errors are transformed into implementation errors, they become even harder to detect because they are transformed and blended into complex implementation structures. Thus, it becomes more troublesome to debug the errors in component-based implementations.

Although design patterns are not formal in nature, design components that have been inspired by design patterns are amenable for formal modeling and analysis. Design patterns are generic design solutions which can be made concrete in an arbitrary number of ways.

Design analysis can assist in discovering bugs in a design early in the development phase and reduce the cost of find-

ing and correcting them downstream. Formal representation and verification techniques are useful for design analysis in that formal representations are more precise, clear, expressive, and unambiguous than informal representations, such as graphical and textual notations. Formal notations can be the basis for verification techniques, such as model checking [3], which can be used to detect errors.

In this paper, we introduce a rigorous modeling and analysis approach to software design composition based on formal specification and automated verification techniques. The approach involves the modeling of design components and their composition, and a framework in which design compositions can be analyzed. We characterize the structural and behavioral aspects, and a specific form of evolution of these components. A case study related to a hypermedia web-based application is presented to illustrate our approach in distributed systems.

In the next section, we provide an overview of our approach and describe a formal model of design component, called design component contract. In section 3, we present a case study of web-based hypermedia application and discuss the analysis of the compositions of design components based on the formal model in logic programming. In the last two sections, we present related work and conclusions.

2 Overview of the Approach

In our approach, we separate the abstract specification from its implementation. The abstract specification contains a formal model of design component, called design component contract, and a category of properties, such as structural, behavioral, hybrid and evolutionary properties. Both the formal models and properties can be implemented, thus, the properties can be verified against the models to detect violations of the properties from the analysis results.

A design component contract includes structural contract and behavioral contract. Structural contract is modeled in predicate logic, whereas the behavioral contract is modeled in Calculus of Communicating Systems (CCS) [24]. Both contracts include three kinds of operations: instantiation, integration and evolution. Since each contract defines the generic information about a design component, the instantiation operation can be used to apply a generic contract in a particular application. The integration operation formally defines how to compose two or more contract to form a new contract. As each design component is not fixed, it often evolves in some restricted ways. The evolution operation formally defines these kinds of changes.

Four different kinds of properties can be analyzed: structural, behavioral, hybrid and evolutionary properties. The structural properties describe the relations of the constructs of each design component, such as connectivity of classes by inheritance or association relations in object-oriented

systems. The structural properties contain atom, consistency, integrity and link properties. The behavioral properties are constraints such as safety, liveness, event ordering, and action sequence of each design component. The behavioral properties consist of global, consistency, concurrency, sequence and occurrence properties. A combination of structural and behavioral properties is a hybrid property. The properties related to system evolution are evolutionary properties, which describe the possible structure changes to adapt new requirements of each design component.

The structural contract is implemented in Prolog. The behavioral contract is implemented in XL that is the model specification language of XMC [27]. Similarly, the structural and evolutionary properties are implemented in Prolog. The behavioral properties are implemented in μ -calculus [21, 29]. The hybrid properties are implemented in Prolog and μ -calculus. XSB Prolog and XMC model checker are used as verification tools to check different properties against their corresponding models. The analysis results show either property verified or counter examples.

In the remainder of this section, we formally define the semantics of the structural and behavioral aspects of a design component contract¹.

2.1 Structural Contracts

The semantics of structural contract is defined by the set of constants, variables, and predicate symbols.

Definition 2.1 (Constant symbols) *The set of constant symbols contains each member of \mathcal{C} (the set of class names), \mathcal{AV} (the set of attribute variable names of all classes), \mathcal{M} (the set of method names of all classes), \mathcal{T} (the set of types), \mathcal{AR} (the set of access rights).*

Definition 2.2 (Variable symbols) *There are five sets of variable symbols ranging over the sets \mathcal{C} , \mathcal{AV} , \mathcal{M} , \mathcal{T} , and \mathcal{AR} , denoted as $V_{\mathcal{C}}$, $V_{\mathcal{AV}}$, $V_{\mathcal{M}}$, $V_{\mathcal{T}}$, and $V_{\mathcal{AR}}$, respectively. In the following we denote with CT (i.e., class terms) as the set $V_{\mathcal{C}} \cup \mathcal{C}$. Similarly, AVT (attribute terms), MT (method terms), TT (type terms), and ART (access right terms) denote the sets $V_{\mathcal{AV}} \cup \mathcal{AV}$, $V_{\mathcal{M}} \cup \mathcal{M}$, $V_{\mathcal{T}} \cup \mathcal{T}$, and $V_{\mathcal{AR}} \cup \mathcal{AR}$, respectively.*

Definition 2.3 (Predicate symbols) *The set of predicate symbols consists of five sets: **Role predicates** (\mathcal{RP}) express the information on the roles of each design component. They define the participants within each design component. **Connection predicates** (\mathcal{CP}) capture the relationships between the roles of each design component. They define how the roles are connected. **Action predicates***

¹We omit the definitions of the instantiation, integration and evolution operations, the modeling rules, and the definitions of structural, behavioral and evolutionary properties. We refer to [8] for details.

(\mathcal{AP}) describe the actions that a role can perform in a design component. **Set predicates** (\mathcal{SP}) depict the information about an arbitrary number of instances of a role. All set predicates are denoted by SPT (i.e., $SPT = \mathcal{SP}$). **Quantification predicates** (\mathcal{QP}) define the way to quantify a set of elements. The sets of role predicates, connection predicates, and action predicates are defined as PT (i.e., $PT = \mathcal{RP} \cup \mathcal{CP} \cup \mathcal{AP}$).

Predicates belonging to \mathcal{RP} , \mathcal{CP} , \mathcal{AP} , \mathcal{SP} , and \mathcal{QP} can be found in [11]. The atoms and literals that can be specified on the basis of the predicate symbols are the corresponding names in each design component.

Definition 2.4 (Structural Contract) *The structural aspect of a design component contract \mathcal{SC} is a tuple $\mathcal{SC} = \langle C, AV, M, T, AR, PS \rangle$, where C is a set of classes in the design component; AV is a set of attributes defined in classes C ; M is a set of methods defined in classes C ; T is a set of types that are used to define the attributes and methods in classes C ; AR is a set of access rights that the attributes and methods can have in a class of C . It provides the mechanism for information hiding. For example, $AR = \{\text{public, protected, private}\}$; and PS is a set of predicate symbols that specify a structural aspect of a design component. These predicate symbols declare the static model of a design component.*

2.2 Behavioral Contracts

In contrast to the structural aspect of a design component contract, the behavioral contract describes the dynamic information, such as the collaboration among the objects participating in the component and the creation of new objects. The behavioral contract is modeled by the collaborations of societies of objects that play different roles and work together to carry out some behavior that is bigger than the sum of the elements. The behavioral contract is essential because the structural contract only captures the static information, but patterns are also characterized by the interactions among the objects and operations. We use process algebras, in particular CCS [24], to define a formal semantic model of behavioral contracts due to their powerful model of behavior and concurrency. The sub-calculus of CCS syntax we consider is shown as follows:

$$P ::= 0 \mid a.P \mid P + P \mid P|P \mid P[f]$$

where terms generated by P are also called processes; the prefixing $a.P$ is sequential composition and the action a range over a nonempty set of actions including a distinguished action τ for unobservable activities; the summation $P + P$ is non-deterministic choice and $\sum_{i \in I} P_i$ is the summation over index set I ; $P|P$ is parallel composition and $\Pi_{i \in I} P_i$ is the parallel composition over index set I ; $P[f]$ is

relabelling where f are relabeling functions preserving observability (i.e., $f^{-1}(\tau) = \{\tau\}$); 0 is nil process that cannot execute any action. We shall often omit the dot of the action prefix, and drop the trailing 0 's from expressions, therefore for example, $a.0 + b.c.0$ is shorthanded to $a + bc$. We refer to [24] for further details of CCS.

Definition 2.5 (Behavioral Contract) *The behavioral aspect of a design component contract \mathcal{BC} is a tuple $\mathcal{BC} = \langle P, IP, OP, IM, OM, IM_I, OM_I, A \rangle$, where P is a finite set of process names; IP is a finite set of input ports attached to a process; OP is a finite set of output ports attached to a process; IM is a finite set of input messages sent to a process, and OM is a finite set of output messages sent from a process; IM_I is the finite set of input messages sent from outside the design component to a process $p \in P$, and OM_I is the finite set of output messages sent outside the design component from a process $p \in P$; A is a set of finite actions that can be performed by a process.*

Definition 2.6 (CCS-Process of Behavioral Contract)

Let $\mathcal{BC} = \langle P, IP, OP, IM, OM, IM_I, OM_I, A \rangle$ be the behavioral contract 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 , $OM(p, i)$ is the set of messages that are sent out by process p when it receives a message i , $P(p, i)$ is the set of processes that are executed by process p when it receives a message i . Then, the CCS-Process $CCS(\mathcal{BC})$ induced by the behavioral contract \mathcal{BC} is defined by introducing, for each process $p \in P$ an equation:

$$BC(p) = \sum_{i \in IM(p)} (\text{in}(p, i).\text{action}(A(p, i)).P(p, i).\text{out}(OM(p, i)).BC(p)).$$

Thus, $CCS(\mathcal{BC}) = (\Pi_{p \in P} BC(p))[f]$.

where $[f]$ is a relabelling operator that is defined as follows.

Definition 2.7 (Relabelling) *Let \mathcal{L} be a set of labels. The relabelling function f is defined by $f : IM - IM_I \rightarrow \mathcal{L}$ and $f : OM - OM_I \rightarrow \mathcal{L}$. For each $i \in (IM - IM_I)$ and $o \in (OM - OM_I)$, if $f(i) = f(o)$, i.e. message i and message o are relabeled to the same label, then these messages synchronize their corresponding processes. This relabelling function can be abbreviated by $b_1/a_1, \dots, b_n/a_n$ so that the f renames a_i to b_i ($f(a_i) = b_i, a_i \neq b_i$), and leaves any other action ($a_j = b_j, j \notin [1..n]$) unchanged. Associated with f is the relabelling operator $[f]$.*

3 Case Study: A Hypermedia Web-Based Application

As the World Wide Web continues to become an important platform for information systems, hypermedia applications keep growing in size and complexity. The design and development of these hypermedia applications have been

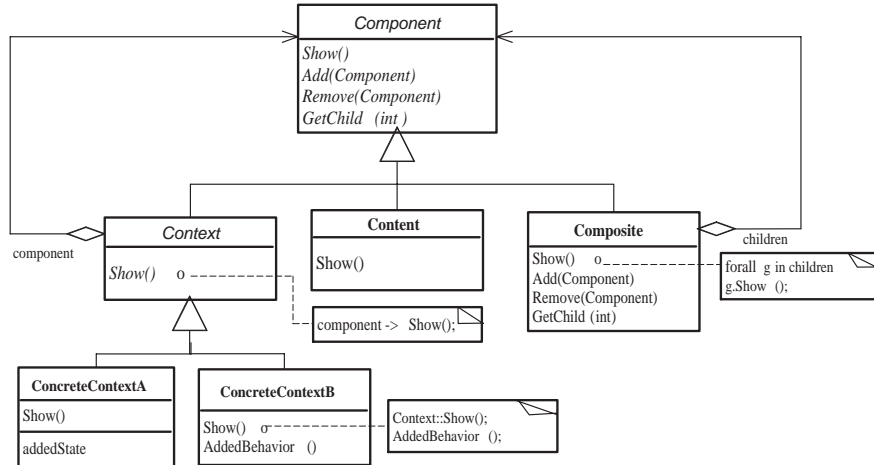


Figure 1. The Navigational Contexts Pattern (Class Diagram)

recognized as a difficult process, especially for large applications [2]. Hypermedia engineering is the application of software engineering practices to the development of hypermedia applications, which include those developed for the World Wide Web.

Hypermedia designers normally do not solve every problem from scratch. They typically reuse the solutions they used previously [28]. It is critical to capture experts' design experience and convey to others. Hypermedia design patterns [28, 18] have been proposed to reuse hypermedia components so that the designer can reason in terms of existing hypermedia structures. Large hypermedia applications often contain many design patterns working in concert to solve complex problems. Many problems are ubiquitous and designers are likely to face them eventually. A catalog of fifty one hypermedia design patterns have been documented in [18]. Discenza [5] has conducted a study of hypermedia design patterns based on thirty four virtual museums all over the world. To illustrate our approach, we analyze the design of a virtual art museum (National Gallery of Art, <http://www.nga.gov/>), which is a complex application containing more than 100,000 objects in the Gallery's collection database [25]. A detailed model and description of this application can be found in [17, 5]. We have chosen to analyze this design with respect to some hypermedia design patterns since this set of patterns illustrates the features of our analysis approach and the kinds of properties that can be checked. We do not show the formalization of all the patterns in this application since the goal of this case study is to show the applicability and accuracy of our approach.

This case study illustrates how to apply our analysis approach in hypermedia applications. In particular, we focused the analysis on the design of a virtual art museum that has many collections of paintings. In a virtual museum, each painting is displayed with some complementary infor-

mation, such as the information about the author, the information about the painting, the time and place of the painting. The paintings in the museum are classified into some categories by different criterion, such as by authors and by time. The user can traverse the paintings through different categories. More specifically, the virtual museum needs to cope with the following design issues: (1) Each painting may have several attributes, such as the graphical presentation of the painting, the textual explanations about its author, some information about the painting and the time and place of the painting, the category to which it belongs. It is required to provide mechanisms for the user to learn each attribute of a painting in an efficient and effective way. (2) It is required to have a history to record the navigation path and allow the user to go backward and forward. (3) The user is able to study the museum with different themes or categories as, e.g., the paintings in 18th century, the paintings of a particular author, the paintings about war, etc. (4) The user needs to have visual knowledge of his/her position in the virtual museum, and can change the position easily.

Hypermedia design patterns, such as the Active Reference, Information on Demand, Navigational Contexts, and Navigational Observer patterns, can be used for the design of this system. In the following, we show the structural and behavioral contracts of the Navigational Context pattern component² and the contracts of the integrations of these components. Consequently, we can show how our formal framework can help the analysis of pattern-based designs.

3.1 Navigational Contexts Pattern

Hypermedia applications usually involve navigating collections of nodes, which may be explored in different orders

²The structural and behavioral contracts of other hypermedia design patterns are presented in [8].

depending on the task the user is performing. For example, collections of paintings may be studied author by author, or explored by different categories, such as nature paintings or architecture paintings. One of the problems is how to organize a collection of nodes such that people can traverse them in different ways as specified previously in the first design issue. The Navigational Contexts pattern solves this problem by separating the context information from the content of a hypermedia component and dynamically attaches different context information to a component [28]. The UML diagram for the Navigational Context pattern is shown in Figure 1. The structural contract is shown in the following.

Example 3.1 Consider the Navigational Contexts pattern described previously. The structural aspect of the design component contract related to this pattern is $SC_{NavCon} = \langle C, AV, M, T, AR, PS \rangle$. The set of classes in the design component is $C = \{Component, Context, Content, Composite, ConcreteContext\}$. The set of attributes defined in classes C is $AV = \{Components, Children\}$. The set of methods defined in classes C is $M = \{Show, Add, Remove, GetChild\}$. The set of types that are used to define the attributes and methods in classes C is $T = \{void\}$. The set of access rights that the attributes and methods can have in a class of C is $AR = \{public, private\}$. The set of predicate symbols is $PS = RP \cup CP \cup AP \cup SP \cup QP$, where

```

RP = { abstractclass(Component), abstractclass(Context),
class(Content), class(Composite), variable(Context, private, Components, Component),
variable(Composite, private, Children, Component), method(Component, public, Show, void),
method(Component, public, Add, void), method(Component, public, Remove, void),
method(Component, public, GetChild, void), method(Content, public, Show, void),
method(Context, public, Show, void), method(Composite, public, Show, void),
method(Composite, public, Add, void), method(Composite, public, Remove, void),
method(Composite, public, GetChild, void) }
CP = { inherit(Component, Content), inherit(Component, Context),
inherit(Component, Composite) }
AP = { invoke(Context, Show, Components, Show) }
SP = { member(ConcreteContext, ConcreteContextSet), member(Child, Children) }
QP = { forall(member(ConcreteContext, ConcreteContextSet), CS),
forall(member(Child, Children), CO) }
CS = { class(ConcreteContext), inherit(Context, ConcreteContext),
method(ConcreteContext, public, Show, void), method(ConcreteContext, public, AddBehavior, void),
invoke(ConcreteContext, Show, Context, Show), invoke(ConcreteContext, Show, ConcreteContext, AddBehavior) }
CO = { invoke(Composite, Show, Child, Show) }

```

The behavior of the Navigational Contexts pattern is defined by the collaborations among the objects participating this pattern. It describes the interaction of these objects and their state changes. The behavioral semantics of this pattern describes how a collection of hypermedia components is aggregated and how the content of each component is attached with its context information (see Figure 2). The behavioral contract of this design component is given next.

Consider the behavior of the Navigational Contexts pattern. The inter-object relationships of this pattern component describe the interactions among the Context, Content and Composite Objects. When an outside request to display a collection of hypermedia components is received by the outermost Context, the Context will first send messages to

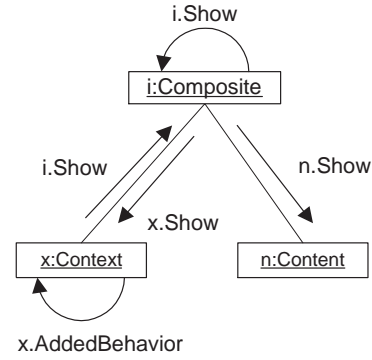


Figure 2. The Navigational Contexts Pattern (Collaboration Diagram)

all inner components to display themselves, and then it will display itself.

Example 3.2 Let $BC_{NavCon} = \langle P, IP, OP, IM, OM, IM_I, OM_I, A \rangle$ be the behavioral contract of Navigational Contexts pattern component and its CCS processes be $CCS(BC_{NavCon})$. The set of processes in the design component is $P = \{Composite, Content, Context\}$. The set of input ports is $IP = \{X2I, X2N, I2X, I2N, Self, Input\}$, where $X2I$ (resp. $X2N, I2X$ or $I2N$) stands for the input port of the Composite (resp. Content, Context or Content) process receiving messages from the Context (resp. Context, Composite or Composite) process, and $Self$ is an input port for message its own process. All messages from outside of the process are received in the input port $Input$. The set of output ports is $OP = \{X2I, X2N, I2X, I2N, Self\}$. The set of input messages is $IM = \{Add, Remove, GetChild, Show\}$. The set of output messages is $OM = \{Show\}$. The set of input messages in the interface is $IM_I = \{Add, Remove, GetChild, Show\}$. The set of output messages in the interface is $OM_I = \{ \}$. The set of actions is $A = \{Add, Remove, GetChild, Show, AddedBehavior\}$. The CCS processes are defined by $CCS(BC_{NavCon})$, where

```

Composite(Name, X2I, I2X, I2N) ::= in(Input, Add).action(Add).Composite(Name, X2I, I2X, I2N) + in(Input, Remove).action(Remove).Composite(Name, X2I, I2X, I2N) + in(Input, GetChild).action(GetChild).Composite(Name, X2I, I2X, I2N) + (in(Input, Show) + in(X2I, Show) + in(Self, Show)).(out(I2X, Show) + out(I2N, Show) + out(Self, Show)).action((Show, Name)).Composite(Name, X2I, I2X, I2N)
Content(Name, X2N, I2N) ::= (in(X2N, Show) + in(I2N, Show)).action(Show(Name)).Content(Name, X2N, I2N)
Context(Name, AddedBehavior, I2X, X2N, X2I) ::= (in(Input, Show) + in(I2X, Show)).(out(X2N, Show) + out(X2I, Show)).action((Show, Name)).action(AddedBehavior).Context(Name, AddedBehavior, I2X, X2N, X2I)
NavigationalContext ::= Composite(aComposite, X2I, I2X, I2N) | Content(aContent, X2N, I2N) | Context(aContext, aBehavior, I2X, X2N, X2I)

```

3.2 Integration

There can be different ways of integrating these pattern components. Figure 3 shows one possible way of the in-

tegration, where the *painting*, *picture*, *author*, *composite*, and *button* classes play the roles of *Component*, *Context*, *Content*, *Content*, *Composite*, and *ConcreteContext*, respectively, in the Navigational Context pattern.

In order to explicitly represent a pattern instance when it is composed with other pattern instances, we extend the UML notation with the tagged pattern notation to depict the pattern and/or participant names associated with a given class, operation or attribute [6]. Each class in Figure 3 is attached with a tag containing the corresponding pattern(s) it participates and the corresponding role(s) it plays in the pattern(s) in the form of “pattern:role”. Symmetrically, each operation (or attribute) is attached with a tag containing the corresponding pattern(s) it participates and the corresponding role(s) it plays in the pattern(s). For instance, the *picture* class participates in two patterns: the Information on Demand pattern and the Navigational Contexts pattern. It plays the role of *NodeAttribute* and *Content* in the two patterns, respectively. The Active Reference, Navigational Contexts, Information on Demand and Navigational Observer patterns are abbreviated as “ActRef”, “NavCon”, “InfoDem” and “NavObs”, respectively.

Example 3.3 Consider the structural contracts of the Active Reference, Navigational Contexts, Navigational Observer and Information on Demand components previously defined. Let the integration of these structural contracts be $SC_{Combine} = \langle C, AV, M, T, AR, PS \rangle$. The set of classes in the design component is $C = \{Page, Presentation, History, AbstractViewer, Viewer, Painting, Reference, Map, Context, Button, Picture, Author, Composite\}$. The set of attributes defined in classes C is $AV = \{History, Subject, Views, Components, references, Children\}$. The set of methods defined in classes C is $M = \{Show, Navigate, Record, Backtrack, Init, Select, Add, Remove, GetChild, Notify, Update, Display, GoTo, ShowButton\}$. The set of types that are used to define the attributes and methods in classes C is $T = \{Painting, Picture, Author, Page, History, AbstractViewer, Reference, void\}$. The set of access rights that the attributes and methods can have in a class of C is $AR = \{public, private\}$. The set of predicate symbols is $PS = RP \cup CP \cup AP \cup SP \cup QP$, which is presented in [11].

Example 3.4 Consider the behavioral contracts of the Active Reference, Navigational Contexts, Navigational Observer and Information on Demand components previously defined. Let the integration of these behavioral contracts be $BC_{Combine} = \langle P, IP, OP, IM, OM, IM_I, OM_I, A \rangle$ and its CCS processes be $CCS(BC_{Combine})$. The set of processes in the design component is $P = \{Picture, Author, Presentation, Page, History, Viewer, Composite, Content, Button, Reference, Painting\}$. The set of input ports is $IP = \{P2N, N2H, H2V, H2N, V2H, X2I, X2N, I2X, I2N, R2C, C2R, Self\}$,

Input. The set of output ports is $OP = \{P2N, N2H, H2V, H2N, V2H, X2I, X2N, I2X, I2N, R2C, C2R, Self\}$. The set of input messages is $IM = \{Init, Select, Display, Navigate, Activate, Record, Backtrack, Add, Remove, GetChild, GoTo, Update, Notify, Show\}$. The set of output messages is $OM = \{Display, Activate, Record, Backtrack, Show, Update\}$. The set of input messages in the interface is $IM_I = \{Init, Select, Navigate, Add, Remove, GetChild, Show, GoTo, Notify\}$. The set of output messages in the interface is $OM_I = \{ \}$. The set of actions is $A = \{Display, Init, Select, Navigate, Activate, Record, backtrack, Add, Remove, GetChild, ShowButton, GoTo, Update, Notify, Show\}$. The CCS processes are defined by $CCS(BC_{Combine})$, where
 InformationOnDemand ::= Node(Picture, P2N) | Node(Author, P2N) | Presentation(Presentation, P2N)
 NavigationalObserver ::= Node(page, N2H, H2N) | History(history, N2H, H2V, V2H, H2N) | Viewer(viewer, V2H, H2V)
 NavigationalContext ::= Composite(composite, X2I, I2X, I2N) | Content(content, X2N, I2N) | Context(button, ShowButton, I2X, X2N, X2I)
 ActiveReference ::= Reference(reference, R2C, C2R) | Component(painting, C2R, R2C)
 Combination ::= InformationOnDemand | NavigationalObserver | NavigationalContext | ActiveReference

3.3 Design Analysis

In this section, we describe our techniques to analyze both the individual design components and the resulting composition. If errors or inconsistencies are found, we need to change the way the components are integrated.

Example 3.5 Consider Prolog implementation of the first consistency property $consistency1(X) :- class(X), abstractclass(X)$, which is checked against the first version of the formal integration contract $SC_{Combine}$ described Example 3.3. The UML diagram of this integration is shown in Figure 3. The verification result is shown in the following.

```
| ?- consistency1(X).
X = painting;
no
| ?-
```

The verification result showed that there was class definition inconsistency related to the *painting* class. In fact, the *painting* class was defined as both an abstract class and a concrete class in the composition of the Prolog descriptions. Since the *painting* class is shared by the Active Reference, Navigational Contexts and Information on Demand pattern components, there should be interactions among these three design components. By taking a close look of these design components, the reason for this inconsistency can be found 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

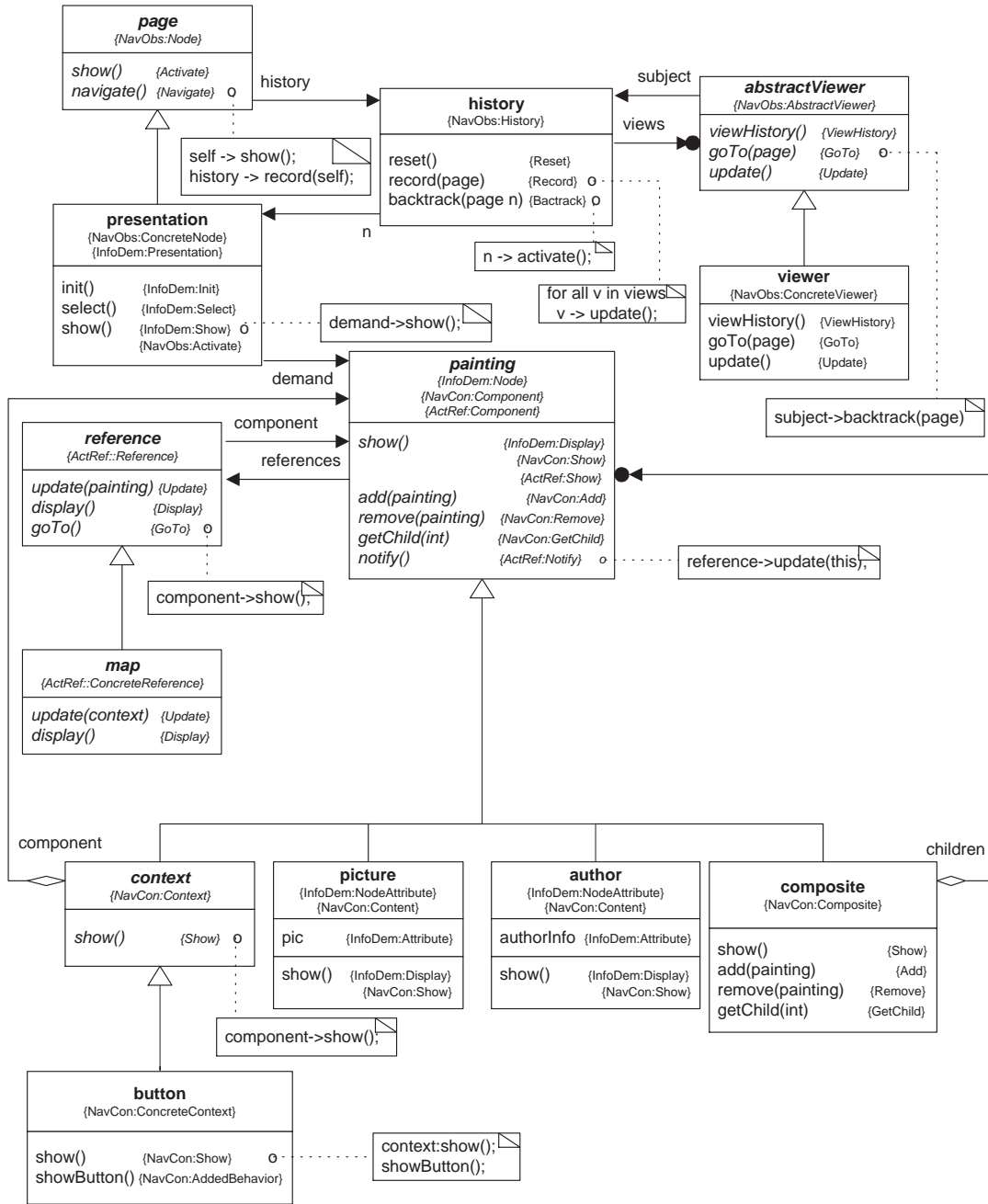


Figure 3. The Version 1 of Design Composition

an abstract class. This means that the *Component* class in the Active Reference pattern cannot be mapped to the *painting* class that has the role of *Component* in the Navigational Contexts pattern and the role of *Node* in the Information on Demand pattern. Therefore, we modify the design composition by mapping it to the *picture* and *author* classes (the role of *Content* in the Navigational Contexts pattern). Thus, the *reference* class (an instance of the *Reference* class in the Active Reference pattern) should have association relationships with the *picture* and *author* classes instead of with the *painting* class in Figure 3. The class diagram of the modified design as the second version of the integration can be found in [11].

In [8], we have defined six kinds of sequence properties: *while*, *eventual*, *possible*, *until*, *unless*, and *then* properties. In the following, we demonstrate the verification of two *then* properties related to this case study. Consider the behavioral contract $BC_{Combine}$ of the integration described in Example 3.4. 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* operations in the *picture* and *author* classes (*sequence1*) and the *show* operation in the concrete context class (*sequence2*), that is the *button* class. These two *show* operations display the content and the context information of a hypermedia component respectively.

Example 3.6 (Then) *The then property that an action a eventually happens and an action b eventually happens after a happens in the integration is denoted by*

$$\mu X.[b]tt \vee ([-]X \wedge [a]tt)$$

The first then property is that the content of a hypermedia component will be eventually displayed when the user changes the current position on an active map. The second then property is that the context of a hypermedia component will be eventually displayed when the user changes the current position on an active map. These two properties are implemented generically in XMC as follows:

```
sequence1(Reference, GoTo, Content, Show) ==
  [rGoTo(Reference, GoTo)] formula1(Content, Show)
  /\ [-] sequence1(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).

sequence2(Reference, GoTo, ConcreteContext, Show) ==
  [rGoTo(Reference, GoTo)] formula2(ConcreteContext, Show)
  /\ [-] sequence2(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).
```

A test Prolog program is written as following:

```
-- import checkit/1 from count.
-- xlc(combine).

test :-
  write('Sequence1'),
  checkit(mck(combination, sequence1(reference, goTo,
    [picture, author], show))),
  write('Sequence2'),
  checkit(mck(combination, sequence1(reference, goTo,
    button, show))).
```

By running the above Prolog program, the model checking results are shown as following:

```
| ?- test.
Sequence1    mck(combination,sequence1(reference, goTo,
[picture, author], show)) is true.
Sequence2    mck(combination,sequence2(reference, goTo,
button, show)) is false.

yes
```

The previous model checking results show that the first property, that the *show* operations in the *picture* and *author* classes are eventually invoked, holds. However, the second property, that the *show* operation in the concrete context class (the *button* class) is eventually invoked, does not hold. Thus, 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 sharing part further down to the concrete context class (*button*) as the third version of the integration shown in Figure 4. The model checking results show that both properties hold this time.

In the following, we show the verification of a hybrid property.

Example 3.7 (Invocation) *Consider the structural contract $SC_{Combine}$ and the behavioral contract $BC_{Combine}$ with its CCS-process $CCS(BC_{Combine})$ of the integration described in Example 3.3 and Example 3.4, respectively. The predicate $invoke(Presentation, Show, Demand, Display)$ is in the set of PS. Thus, eventually an action $action(Show) \in A$ occurs and eventually an action $action(Display) \in A$ will occur after $action(Show)$, which is denoted by*

$\mu X.[action(Show)]tt \vee ([-]X \wedge [action(Display)]tt)$ *in μ -calculus, which is a sequence property and can be implemented in XMC as following:*

```
sequence3(Presentation, Show, Demand, Display) ==
  [pShow(Presentation, Show)] formula1(Demand, Display)
  /\ [-] sequence3(Presentation, Show, Demand, Display).
```

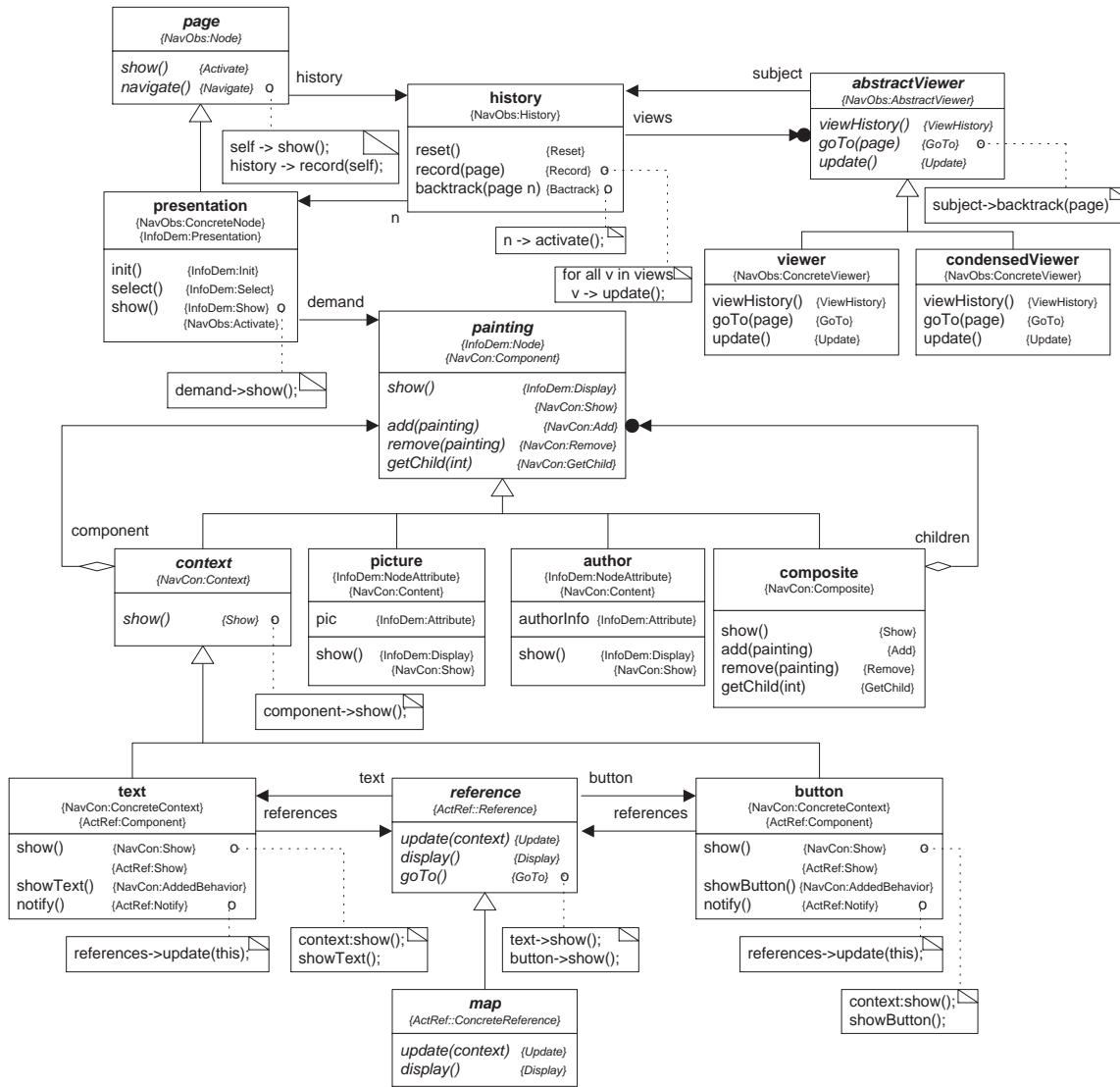


Figure 4. The Version 3 of Design Composition

```

formula1(Demand, Display) +=
  <display(Demand, Display)> tt
  \ / form3(Demand, Display)
  \ / [-] formula3(Demand, Display).

form3(Demand, Display) +=
  <display(Demand, Display)> tt
  \ / [-{pShow(_,_) }] form3(Demand, Display).

```

A test Prolog program is written as following:

```

:- import checkit/1 from count.
:- xlc(combine).

test :-
  invoke(presentation, show, demand, display),
  write('Sequence3'),
  checkit(mck(combination, sequence3(presentation,
    show, demand, display))).

```

where the test result is true if both the structural property *invoke(presentation, show, demand, display)* and the behavioral property *sequence3(presentation, show, demand, display)* hold. Since the model checking of behavioral properties can be described as a Prolog rule (*checkit1*), this test program combines the checking of a structural property and a behavioral property in Prolog. By running the above Prolog program, the results are shown as following:

```

| ?- test.
Sequence3      mck(combination,sequence3(presentation,
show, demand, display)) is true.

yes

```

The results show that both the structural property and the behavioral property hold.

4 Related Work

The notion of contracts in software development is attributed to Meyer [23]. Another contribution, the OO-contracts of Helm *et al.* [19] focus on specifying the behavior and interactions between objects in a system. Helm *et al.* noticed that the behavior of an object could not be inferred from its interface, leading to design and reuse problems. Contracts formalize the behavioral relationship between objects and define a set of participants and their obligations. In [1], Beugnard *et al.* propose a four-level contract for components to increase trust. In this paper, we defined a formal model of design component based on contract and a rigorous analysis approach to software design composition based on automated verification techniques.

Keller *et al.* [20] described a methodical approach to design composition which was illustrated as a process within a four-dimensional design space. They characterized a special kind of component, called design component, and discussed a development process to compose these components at the design level and generate source code frames or executable code. Although our approach is also in the area

of software composition, it focuses on the formal, declarative, and property-based aspects of design composition.

Model checking implicit-invocation systems has been investigated in [15]. Implicit-invocation systems were modeled by the structure elements including components, event types, shared variables, event bindings, event delivery policies and concurrency models. A run-time state model was also constructed with the mechanisms that handle event announcement, event buffering, and method invocation and the mechanisms that implement event dispatch and event delivery policy. This modeling process is highly domain-specific. Modeling techniques for one class of software systems may be completely inappropriate for another. Our work on modeling design component contracts can be seen as another domain-specific model checking framework.

Property patterns [12, 13] have been proposed to provide taxonomy of properties written in LTL [22], CTL [4] and QRE [26]. Each property pattern is defined in terms of its scope, which is the extent of the program execution over which the pattern should hold. Five basic kinds of scopes, global, before, after, between and after-until, have been identified. There are three property patterns: occurrence patterns, ordering patterns and compound patterns. This work provided taxonomy of properties in general. This taxonomy may help users to understand and use these properties. However, besides general properties, we address specific properties related to the domain of design component such as evolutionary and hybrid properties.

5 Conclusions

In this paper, we have introduced an overview of a formal model of design component based on contract and a rigorous analysis approach to software design composition based on automated verification techniques. The definition and analysis of design component contract are based on logic programming. Discovering composition errors at the design level is typically much easier than at the code level because a small piece of design may be mapped to thousands of lines of implementation code. The design errors may be hidden in complex implementation structures and are very costly to analyze or detect. Further, at the implementation stage they are very costly to modify. In this paper, we provide a case study to illustrate our approach on detecting design errors in the compositions of design components. Our work includes formal definitions of design components and their compositions in general. Instances of a design component can be derived from these formal definitions. Their compositions can be analyzed by checking the properties of these 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, 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. Fifth, contracts were able to capture the complex design component topologies and interactions and could be used to analyze pattern-based designs.

References

- [1] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.
- [2] M. Bieber and F. Vitali. Toward Support for Hypermedia on the World Wide Web. *IEEE Computer*, 30(1):62–70, January 1997.
- [3] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
- [4] 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.
- [5] A. Discenza. Design Patterns for WWW Museum Hypermedia. *Technical Report 99.4*, Politecnico di Milano, 1999.
- [6] J. Dong. Representing the Applications and Compositions of Design Pattern in UML. *Proceedings of the ACM Symposium on Applied Computing (SAC)*, Melbourne, Florida, USA, pages 1092–1098, March 2003.
- [7] J. Dong. Towards A Formal Design Component Framework. *Proceedings of the STEP Workshop on Software Development Methodologies of Distributed Systems*, Amsterdam, The Netherlands, Sept. 2003.
- [8] J. Dong. Design Component Contracts: Model and Analysis of Pattern-Based Composition. *Ph.D. Thesis, Computer Science Department, University of Waterloo*, June 2002.
- [9] J. Dong, P. Alencar, and D. Cowan. A Behavioral Analysis Approach to Pattern-Based Composition. *Proceedings of the 7th International Conference on Object-Oriented Information Systems (OOIS)*, Springer Verlag, Calgary, Canada, pages 540–549, August 2001.
- [10] J. Dong, P. Alencar, and D. Cowan. A Formal Framework for Design Component Contracts. *Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI)*, pages 53–60, October 2003.
- [11] J. Dong, P. Alencar, and D. Cowan. Automating the Analysis of Design Component Contracts. *Technical Report UTDCS-01-04*, Computer Science Department, University of Texas at Dallas, 2004.
- [12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15, March 1998.
- [13] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. *Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA*, May 1999.
- [14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.
- [15] D. Garlan and S. Khersonsky. Model Checking Implicit-Invocation Systems. *Proceedings of the 10th International Workshop on Software Specification and Design (IWSSD)*, pages 23–30, November 2000.
- [16] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, (Leavens G. and Sitaraman M., eds.) Cambridge University Press, pages 47–67, 2000.
- [17] D. M. Germán. Hadez: A Framework for the Specification and Verification of Hypermedia Applications. *Ph.D. Thesis, Computer Science Department, University of Waterloo*, 2000.
- [18] D. M. Germán and D. D. Cowan. Towards a Unified Catalog of Hypermedia Design Patterns. *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, January 2000.
- [19] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, USA, pages 169–180, October 1990.
- [20] R. K. Keller and R. Schauer. Design Components: Towards Software Composition at the Design Level. *Proceedings of the 20th International Conference on Software Engineering*, pages 302–311, 1998.
- [21] D. Kozen. Results on the Propositional μ -calculus. *Theoretical Computer Science*, 27:333–354, 1983.
- [22] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1991.
- [23] B. Meyer. Applying “design by contract”. *IEEE Computer*, pages 40–51, October 1992.
- [24] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
- [25] NGA. The National Gallery of Art. Available from <http://www.nga.gov/help/help.htm>, 2002.
- [26] K. Olender and L. Osterweil. Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, March 1990.
- [27] Y. Ramakrishna, C. Ramakrishnan, I. Ramakrishnan, S. Smolka, T. Swift, and D. Warren. Efficient Model Checking Using Tabled Resolution. *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, Haifa Israel, LNCS1243, Springer Verlag, pages 143–154, July 1997.
- [28] G. Rossi, D. Schwabe, and A. Garrido. Design Reuse in Hypermedia Applications Development. *Proceedings of the ACM International Conference on Hypertext*, pages 57–66, April 1997.
- [29] C. Stirling. An Introduction to Modal and Temporal Logics for CCS. *Lecture Notes in Computer Science 491*, Springer Verlag, pages 1–20, 1991.