SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **36**:27–71
Published online 15 September 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.681

Automating the analysis of design component contracts

Jing Dong[1,*,†], Paulo S. C. Alencar[2] and Donald D. Cowan[2]

[1] *Department of Computer Science, University of Texas at Dallas, Dallas, TX 75083, U.S.A.*
[2] *School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1*

SUMMARY

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 an overview of 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 has been presented to illustrate our approach in distributed systems. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: component-based software design; contract; design pattern

1. INTRODUCTION

Component-based software development focuses on building large software systems by integrating existing software components [1–3]. It is a promising solution to some of the problems that designers, developers and integrators face when building their systems. 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 [3]. For successful component-based software development several prerequisites are needed, including a technology base in the form of component models, a selection of commercial off-the-shelf components, integration techniques, run-time environments, development methods and development tools.

*Correspondence to: Jing Dong, Department of Computer Science, University of Texas at Dallas, Dallas, TX 75083, U.S.A.
†E-mail: jdong@utdallas.edu

Copyright © 2005 John Wiley & Sons, Ltd.

Received 4 May 2004
Revised 14 January 2005
Accepted 19 January 2005

Design patterns [4] capture the distilled experience of expert designers. Patterns are not invented, rather they are 'excavated' from existing systems. The excavating process involves the extraction of designs from a number of systems, looking for patterns in designs across those systems. 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 [2], 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 [2,5–7]. 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 to formal modeling and analysis. Design patterns are generic design solutions which can be made concrete in an arbitrary number of ways. The focus on design components is important because one of the goals of our work is to detect errors as early as possible in the development process by reasoning about the properties at the design level and reducing the cost of finding and correcting these errors in concrete software components.

Design analysis can assist in discovering bugs in a design early in the development process and reduce the cost of finding 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 [8], which can be used to detect errors.

In this paper, we introduce an overview of 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.

Our main goal is to provide a systematic approach for a software designer to model and analyze component integration during the design phase, the early planning stage of the software lifecycle. The approach includes a process of representing, instantiating and integrating design components and analyzing their composition. With this approach, the designer can not only model the design component precisely, unambiguously and expressively, but also detect the interactions between components and correct design errors before implementation. The approach helps the designer weigh the tradeoffs of design decisions and choose appropriately.

Copyright © 2005 John Wiley & Sons, Ltd.

In the next section, we provide an overview of our approach. In Section 3, we describe a formal model of a design component, called a design component contract. In Section 4, we present a case study of Web-based hypermedia application and discuss the analysis of the compositions of design components based on a formal model in logic programming. In the last two sections, we present related work and conclusions.

2. OVERVIEW OF THE APPROACH

In this section, we present an overview of our approach and outline the general ideas behind our formal models. Figure 1 gives a high-level overview of our approach. We separate the abstract specification from its implementation. The abstract specification contains a formal model of a design component, called a design component contract, and a category of properties, such as structural, behavioral, hybrid and evolutionary properties. Both the formal models and the properties can be implemented, thus the properties can be verified against the models to detect violations of the properties from the analysis results.

The upper part of Figure 1 shows the abstract specifications of design component contracts. A design component contract includes a structural contract and a behavioral contract. A structural contract is modeled in predicate logic, whereas a behavioral contract is modeled in the Calculus of Communicating Systems (CCS) [9]. 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 contracts 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. The abstract specifications of the formal models of design components are described, in detail, in Section 3.

Four different kinds of properties can be analyzed: structural, behavioral, hybrid and evolutionary properties. The structural properties contain atom, consistency, integrity and link properties. 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.

The lower part of Figure 1 describes the implementation of our approach. The structural contract is implemented in Prolog. The behavioral contract is implemented in XL that is the model specification language of XMC [10]. Similarly, the structural and evolutionary properties are implemented in Prolog. The behavioral properties are implemented in μ-calculus [11,12]. 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. In this way, the CCS and μ-calculus specifications are transformed into logic programs by XMC for analysis. The analysis results either show that the property is verified or provide counter examples by online justification in XMC [13]. When errors are found, the designer is able to pin-point the cause of error by the counter examples provided by XMC. Thus, the designer can re-model the design to correct the error.

There can be different ways to implement design component contracts. The lower part of Figure 1 shows only one possible implementation. Other tools that support the verification of the specification based on predicate logic and process algebra are also candidates.

Copyright © 2005 John Wiley & Sons, Ltd.

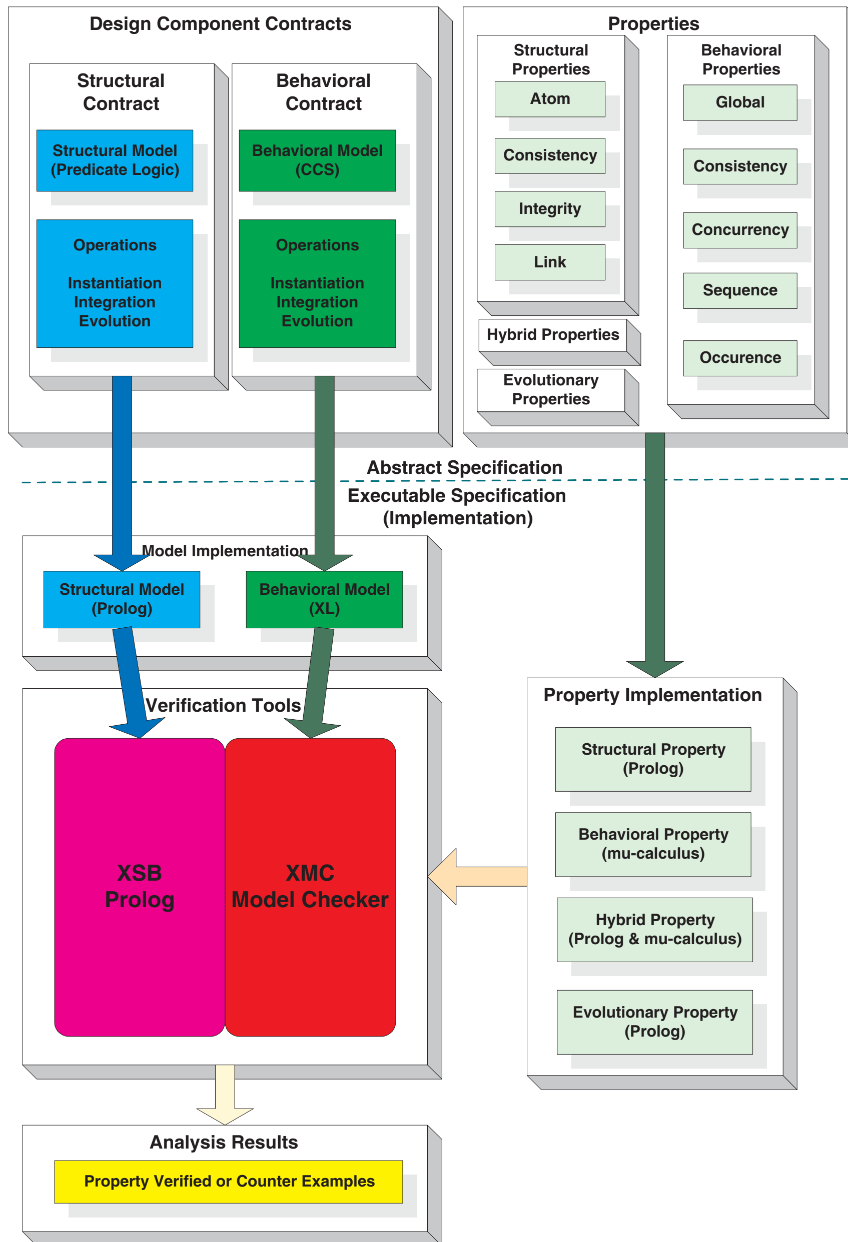

Figure 1. High-level overview.

Copyright © 2005 John Wiley & Sons, Ltd.

3. DESIGN COMPONENT CONTRACTS

A design component contract is a formal model of a design component and its operations, such as instantiation, integration and evolution. The structural and behavioral aspects of a contract are defined based on logic programming and process algebra, respectively. This model of design component contract, together with the property specifications, form the foundation for rigorous analysis of the composition of these components. This analysis process is supported by tools with model-checking and theorem-proving techniques.

A design component contract consists of several parts.

- **Name:** each design component contract has a unique name.
- **Structural contract:** the structural aspect of a design component is modeled based on a logic program. A structural contract is defined by constant symbols, variable symbols and predicate symbols. A model of a structural contract is instantiated, evolved and integrated with other contracts in a particular application.
- **Behavioral contract:** the behavioral aspect of a design component is modeled based on a process algebra. A behavioral contract defines the interactions among the objects as processes and their communications. The integration and evolution of behavioral contracts are defined based on the interface of each contract.
- **Operation:** an operation describes the tasks a contract can perform on itself or other contracts. There are three operations as follows.
  - **Instantiation:** as each contract is a model of a design component, the application of the component requires instantiating these generic definitions with application-dependent information.
  - **Evolution:** each component is modeled with changes in mind in that it describes an expected evolution path in terms of addition or removal of certain parts of the component.
  - **Integration:** the composition of a design component contract is defined based on different aspects of each individual component. The composition of structural contracts is accomplished by set union, whereas that of behavioral contracts is accomplished by parallel composition.
- **Property:** each design component possesses certain characteristic properties. Without these properties, the component may lose its identity. These properties are also supposed to hold when the component is instantiated, changed or integrated with other components. The properties are classified based on the different aspects related to structure, behavior and evolution.

In order to provide a semantic foundation for our design component contract model and property mode, we represent the structural aspect of a design component contract as clauses in a normal logic program [14] and the behavioral aspect of a design component contract by a process algebra, the CCS [9]. Meanwhile, we express structural properties by logic rules and behavioral properties by μ-calculus [11]. This allows us to check the properties of the compositions of design components formally.

Copyright © 2005 John Wiley & Sons, Ltd.

In the remainder of this section, we formally define the semantics of the structural and behavioral aspects of a design component contract and the integration operation‡.

3.1. Structural contracts

The semantics of a design component contract is defined by a set of constants, variables and predicate symbols. We then define the rules that can be expressed in the language.

Definition 3.1 (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) and $\mathcal{AR}$ (the set of access rights).

Definition 3.2 (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 3.3 (Predicate symbols). The set of predicate symbols consists of five sets as follows.

- **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** ($\mathcal{AP}$) 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 $\mathcal{QP}$ are listed in Appendix A. The atoms and literals that can be specified on the basis of the predicate symbols are the corresponding names in each design component.

The structural aspect of a design component contract is defined as follows.

Definition 3.4 (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, and 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, $AR = \{public, protected, private\}$ and PS is a set of predicate symbols that specify a structural aspect

‡We omit the definitions of the instantiation and evolution operations, the modeling rules and the definitions of structural, behavioral and evolutionary properties. We refer to [15] for details.

Copyright © 2005 John Wiley & Sons, Ltd.

of a design component. These predicate symbols, which are defined in Appendix A declare the static model of a design component.

3.2. Structural integration

Definition 3.5 (Integration). Let $SC_1 = \langle C_1, AV_1, M_1, T_1, AR_1, PS_1\rangle$ and $SC_2 = \langle C_2, AV_2, M_2, T_2, AR_2, PS_2\rangle$ be the structural contracts of two design components. Then, the integration of SC_1 and SC_2 is denoted by $SC = \langle C, AV, M, T, AR, PS\rangle$, where the integration operation is defined by two mapping functions: $\delta_1 : C_1 \times AV_1 \times M_1 \times T_1 \times AR_1 \times PS_1 \rightarrow C \times AV \times M \times T \times AR \times PS$ and $\delta_2 : C_2 \times AV_2 \times M_2 \times T_2 \times AR_2 \times PS_2 \rightarrow C \times AV \times M \times T \times AR \times PS$, where $C = \delta_1(C_1) \cup \delta_2(C_2)$, $AV = \delta_1(AV_1) \cup \delta_2(AV_2)$, $M = \delta_1(M_1) \cup \delta_2(M_2)$, $T = \delta_1(T_1) \cup \delta_2(T_2)$, $AR = \delta_1(AR_1) \cup \delta_2(AR_2)$, and $PS = \delta_1(PS_1) \cup \delta_2(PS_2)$. For each predicate symbol $a(x_1, \ldots, x_i, \ldots, x_n) \in PS_1$ and $b(y_1, \ldots, y_i, \ldots, y_n) \in PS_2$ where $1 \leq i \leq n$, if $\delta_1(a(x_1, \ldots, x_i, \ldots, x_n)) = \delta_2(b(y_1, \ldots, y_i, \ldots, y_n))$ (i.e. $a(\delta_1(x_1), \ldots, \delta_1(x_i), \ldots, \delta_1(x_n)) = b(\delta_2(y_1), \ldots, \delta_2(y_i), \ldots, \delta_2(y_n)) \Rightarrow \delta_1(x_i) = \delta_1(y_i), 1 \leq i \leq n$), then the predicate symbols $\mathrm{a}(x_1, \ldots, x_i, \ldots, x_n)$ and $\mathrm{b}(y_1, \ldots, y_i, \ldots, y_n)$ are mapped to the same predicate symbol that is an overlapping part of the two design components.

3.3. 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 implement some behavior that is larger 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 have chosen process algebras for defining a formal semantic model of behavioral contracts because they represent a powerful approach to modeling of behavior and concurrency. Process algebras allow for hierarchical description of processes and they are amendable to verification, analysis and compositional reasoning. The particular process algebra we have chosen for modeling the semantics of behavioral contracts is CCS [9]. CCS is a powerful modeling language and it is well supported with model-checking tools, such as XMC [10].

The sub-calculus of CCS syntax we consider is shown as follows:

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

where: terms represented by P are also called processes; the prefixing $a.P$ is sequential composition and the action a range over a non-empty 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 relabeling where f are relabeling functions preserving observability (i.e. $f^{-1}(\tau) = \{\tau\}$); and 0 is a nil process that cannot execute any action. We often omit the dot of the action prefix, and drop the trailing 0's from expressions, therefore, for example, $a.0 + b.c.0$ becomes $a + bc$. CCS is mainly used to model two-process synchronization in this paper. It can also be used to model broadcast messages by connecting each receiver with the sender through a channel. When the

Copyright © 2005 John Wiley & Sons, Ltd.

sender needs to broadcast a message, it may send the message to all of its receivers through their corresponding channels. We refer to [9] for further details of CCS.

Definition 3.6 (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 $p \in P$, and A is a set of finite actions that can be performed by a process.

Definition 3.7 (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(BC) induced by the behavioral contract BC is defined by introducing, for each process $p \in P$ an equation:

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

Thus,

$$\text{CCS}(BC) = (\Pi_{p \in P} BC(p))[f]$$

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

Definition 3.8 (Relabeling). Let $\mathcal{L}$ be a set of labels. The relabeling 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 relabeling function can be abbreviated by $b_1/a_1, \ldots, b_n/a_n$ so that f renames a_k to b_k ($f(a_k) = b_k, a_k \neq b_k, k \in [1 \ldots n]$), and leaves any other action ($a_j = b_j, j \in [1 \ldots n]$) unchanged. Associated with f is the relabeling operator $[f]$.

4. CASE STUDY: A HYPERMEDIA WEB-BASED APPLICATION

The World Wide Web and its associated hypermedia applications continue to grow in size and complexity. Web-based applications range from simple personal home sites composed of a few pages to corporate Web sites consisting of thousands of pages. The design and development of these hypermedia applications has been recognized as a difficult process, especially for large applications [16]. Hypermedia engineering is the application of software engineering practices to the development of hypermedia applications, which include those developed for the World Wide Web.

One of these efforts is the creation of design methodologies, such as the Relational Management Methodology (RMM) [17], the Hypermedia Design Model (HDM) [18], the Object-Oriented Hypermedia Design Model (OOHDM) [19], and the Enhanced Object-Relationship Model (EORM) [20]. These methodologies are guidelines to be followed during the design process. They also

Copyright © 2005 John Wiley & Sons, Ltd.

specify the characteristics of the deliverables, which are created at each of their stages. These products are not usually formally specified, in the sense that they do not have a formal syntax or semantics, and they are not required to pass validity tests.

Hypermedia designers normally do not solve every problem from scratch. They typically reuse the solutions they used previously [21]. It is critical to capture expert design experience and convey it to others. Hypermedia design patterns [21,22] have been proposed to reuse hypermedia components so that the designer can reason in terms of existing hypermedia structures. Patterns help expert designers to communicate their design decisions. Meanwhile, it becomes easier for novice designers to learn quickly through the comprehensible pattern format. Although there is not yet consensus on design methodologies in the hypermedia community, patterns transcend this dimension in that they capture the essential aspects of design problems and their solutions.

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 51 hypermedia design patterns has been documented in [22]. Discenza [23] has conducted a study of hypermedia design patterns based on 34 virtual museums all over the world. Through these research efforts, many hypermedia design patterns have been documented and categorized, such as the Active Reference pattern, the Clustering pattern, the Collector pattern, the Glue pattern, the Information on Demand pattern, the landmark pattern, the Navigational Contexts pattern, the Navigational Observer pattern, the Neighborhood pattern, the News pattern and the Session pattern.

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 from the Gallery's collection database [24]. A detailed model and description of this application can be found in [23,25]. We have chosen to analyze this design with respect to some hypermedia design patterns. 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.

In the following sections, we first describe an overview of a hypermedia application and show some of the issues related to this application. We then discuss a pattern-based design to address the issues previously described, and provide the contracts of all design components used in this design. Later, we show integration of these design components. In Section 5 we analyze different kinds of properties defined in [15].

4.1. An overview of the application

This case study illustrates how to apply our analysis approach in hypermedia applications. In particular, we focus 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 information, such as information about the author and the painting, as well as the time and place of the painting. The paintings in the museum are also classified into some categories by different criteria, such as author, time period and subject. The user can traverse the paintings through these 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 the author, information about the painting, the time and place of

Copyright © 2005 John Wiley & Sons, Ltd.

the painting, and the category to which it belongs. Mechanisms are required to enable the user to learn about each attribute of a painting efficiently and effectively.
2. A history of the navigation path is also necessary so that the user can move backward or forward from the current position.
3. The user is able to study the museum using different themes or categories such as the paintings of the 18th century, the paintings of a particular author, or the paintings about a theme such as war.
4. The user needs to have visual knowledge of their position in the virtual museum, with the ability to change the position easily.

4.2. A pattern-based design

The design issues presented in the previous section are ubiquitous and can be found in many virtual museum applications [23]. Several hypermedia design patterns, such as the Navigational Contexts, the Navigational Observer patterns, the Active Reference and the Information on Demand, can be used for the design of this system. In the following, we show the structural and behavioral contracts of the Navigational Context pattern and the Navigational Observer pattern components[§] and the contracts related to the integration of these pattern components. Consequently, we can show how our formal framework can help with the analysis of pattern-based designs.

4.2.1. Navigational contexts pattern

Hypermedia applications usually involve navigating collections of nodes, which may be explored in different orders depending on the task the user is performing. For example, collections of paintings may be studied by author, or explored by different categories, such as nature or architecture. One of the problems is how to organize a collection of nodes such that people can traverse them in different ways. 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 [21]. The UML diagram for this integration is shown in Figure 2.

The intent of this pattern is to group hypermedia elements with similar attributes and attach context-related guidelines and relationships to each group for better presentation and comprehension. The same hypermedia element is also allowed to look different when perceived within different groups (contexts). For example, a collection of paintings can be explored author by author or subject by subject. It is desirable to provide the user with different kinds of feedback in different contexts for a given painting.

Since every Web page is independent, information common to several pages may be repeated, presenting a potential consistency problem. When the information in one page is changed, more specifically, all of its counterparts in other pages should be modified accordingly. The separation of the content information from the context information is important because it solves the consistency problem described previously by considering the information common to several pages as content and

[§]The structural and behavioral contracts of other hypermedia design patterns are presented in [15].

Copyright © 2005 John Wiley & Sons, Ltd.

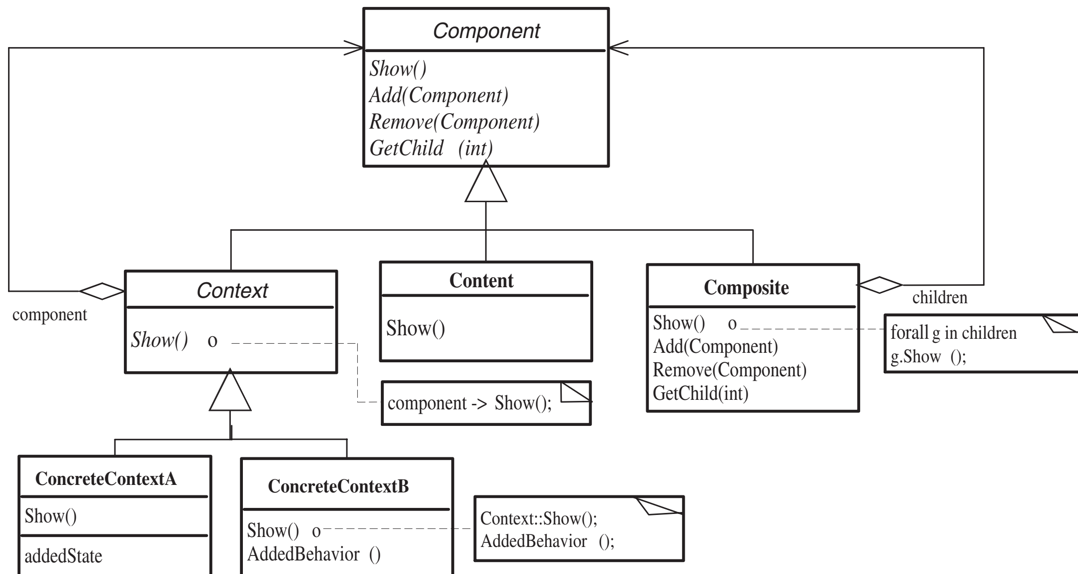

Figure 2. The Navigational Contexts pattern (class diagram).

all other information as context. Thus, the switch of contexts can be easily achieved by dynamically attaching the corresponding context to the content without affecting it.

In hypermedia applications, the navigational structure is normally embedded as hyperlinks in the files. Thus, it is difficult to find and modify these navigational links. The Navigational Contexts pattern can be also used to solve this navigation problem by isolating context-related navigational links from the content so that they are easy to change.

Based on Definition 3.4, the structural contract of the Navigational Contexts pattern component can be defined as follows.

Example 4.1. Consider the Navigational Contexts pattern described previously. The structural aspect of the design component contract related to this pattern is $SC_{NavCon} = \langle C, AV, M, T, AR, PS \rangle$, where:

- 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 that specify a structural aspect of a design component is $PS = RP \cup CP \cup AP \cup SP \cup QP$, where

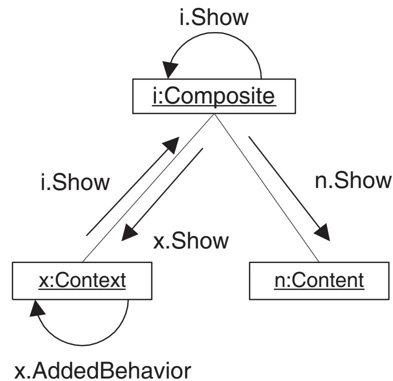

Figure 3. The Navigational Contexts pattern (collaboration diagram).

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 in this pattern. It describes the interaction of these objects and their state changes. The behavioral semantics of this pattern describe how a collection of hypermedia components is aggregated and how the content of each component is attached with its context information (see Figure 3). The behavioral contract based on Definition 3.6 is presented next.

Copyright © 2005 John Wiley & Sons, Ltd.

Example 4.2. 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 all inner components to display themselves, and then it will display itself. 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 let its CCS processes be $\mathrm{CCS}(BC_{NavCon})$, where we have the following.

- 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 $X2I$ (respectively $X2N$, $I2X$ or $I2N$) stands for the input port of the Composite (respectively Content, Context or Content) process receiving messages from the Context (respectively 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 $\mathrm{CCS}(BC_{NavCon})$, 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)

4.2.2. *Navigational Observer pattern*

It is common for many hypermedia applications to record the history of navigation. This history can be perceived in many different ways. Any change of the history should be reflected in each view of the history. The Navigational Observer pattern solves this problem by having different views of a given hypermedia entity. These views are synchronized with the hypermedia entity. This pattern is quite similar to the Observer pattern [4]. The difference is that its intention is for viewing history.

Navigational Observer decouples hypermedia components from the navigational history and the history from its interface (viewers). This decoupling allows different viewers or various

Copyright © 2005 John Wiley & Sons, Ltd.

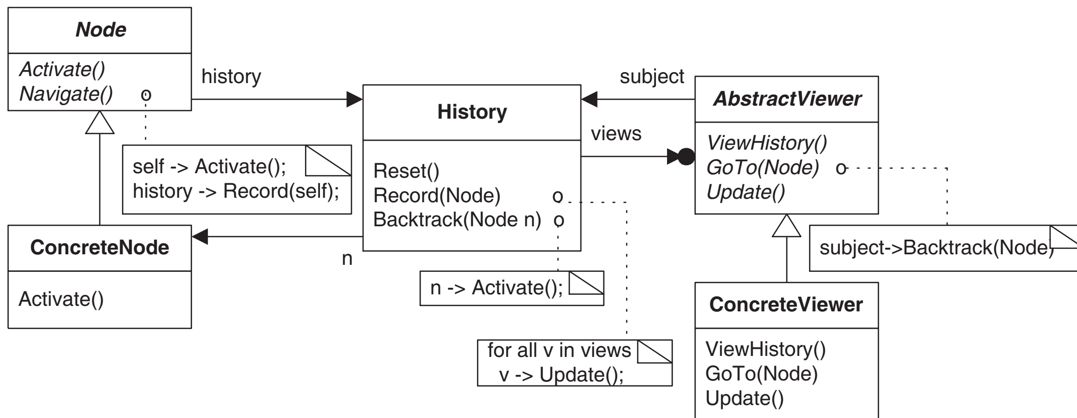

Figure 4. The Navigational Observer pattern (class diagram).

implementations of the history without affecting the corresponding hypermedia components. The concept of history is very common in hypermedia applications so that it should be applied in different applications uniformly.

When the name, hyperlink or Uniform Resource Locator (URL) of a resource changes, all the links pointing to that resource have to be updated. This problem exists at both the global level, where the author does not have control over the documents pointing to the local information from an external site, and at the local level. The Navigational Observer pattern can solve this problem as well. The separation of the subjects and the views attached to them allows the changes of a subject to be reflected on all views attached to this subject automatically. In fact, a navigational history typically consists of a list of hyperlinks. Any change of a navigational history is revealed to each viewer of this history.

The class diagram for the structural aspect of this pattern is shown in Figure 4. The *Node* class and its descendants are the hypermedia components. When a user visits a node, the node activates itself and lets the *History* class record itself in the history list. The *History* class provides mechanisms to manage the history list as well as notify all of its viewers to update their presentations. The *Reset* operation in the *History* class is used to reset the history. The *Backtrack* operation is used to trace back to a particular point in the history. The user can also choose to navigate inside the current history list by going to a selected node in the history. More specifically, the *GoTo* operation in the Viewer class is invoked, which will invoke the *Backtrack* operation in the *History* class. The *ViewHistory* operation presents the history in a particular way. The *Update* operation can change the presentation of the history when it is changed.

The collaboration diagram is shown in Figure 5. When the user is visiting a node, the *Activate* operation is invoked to display the corresponding hypermedia component. Meanwhile, a message *h.Record* is sent to the History object to append the current hypermedia component to the end of the history list. The messages *v.Update* are then sent to all viewers to update their presentations. When the

Copyright © 2005 John Wiley & Sons, Ltd.

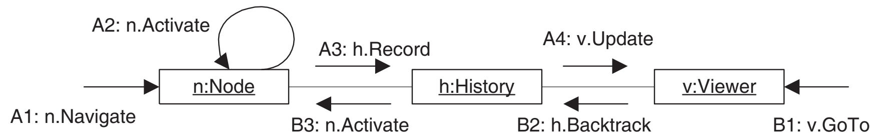

Figure 5. The Navigational Observer pattern (collaboration diagram).

user wants to go back to any point in the history, they can use the *Backtrack* operation which in turn sends a message *n.Activate* to display the corresponding hypermedia component.

Example 4.3. Consider the Navigational Observer pattern described previously. The structural aspect of the design component contract related to this pattern is $SC_{NavObs} = \langle C, AV, M, T, AR, PS \rangle$, where we have the following.

- The set of classes in the design component is C = {*Node, ConcreteNode, History, Abstract Viewer, ConcreteViewer*}.
- The set of attributes defined in classes C is AV = {*Subject, Views, Histories*}.
- The set of methods defined in classes C is M = {*Activate, Navigate, Reset, Record, Backtrack, ViewHistory, GoTo, Update*}.
- The set of types that are used to define the attributes and methods in classes C is T = {*Node, History, AbstractViewer, 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 that specify a structural aspect of a design component is $PS = RP \cup CP \cup AP \cup SP \cup QP$, where

RP = {
abstractclass(Node),
abstractclass(AbstractViewer),
class(History),
variable(Node, private, Histories, History),
variable(AbstractViewer, private, Subject, History),
variable(History, private, Views, AbstractViewer),
method(Node, public, Activate),
method(Node, public, Navigate),
method(History, public, Reset),
method(History, public, Record),
method(History, public, Backtrack),
method(AbstractViewer, public, ViewHistory),
method(AbstractViewer, public, GoTo, Node),
method(AbstractViewer, public, Update)
}

CP = {
}
AP = {
invoke(Node, Navigate, self, Activate),
invoke(Node, Navigate, Histories, Record),
invoke(History, Backtrack, n, Activate),
invoke(AbstractViewer, GoTo, Subject, Backtrack),
}
SP = {
member(ConcreteNode, ConcreteNodeSet),
member(ConcreteViewer, ConcreteViewerSet),
member(V, Views)
}

```
QP = {
forall(member(ConcreteNode, ConcreteNodeSet), CS),
forall(member(ConcreteViewer, ConcreteViewerSet), CO),
forall(member(V, Views), CI)
}
CS = {
class(ConcreteNode),
inherit(Node, ConcreteNode),
method(ConcreteNode, public, Activate)
}
```

```
CO = {
class(ConcreteViewer),
inherit(AbstractViewer, ConcreteViewer),
method(ConcreteViewer, public, ViewHistory),
method(ConcreteViewer, public, GoTo),
method(ConcreteViewer, public, Update)
}
CI = {
invoke(History, Record, V, Update)
}
```

Example 4.4. Consider the behavior of the Navigational Observer pattern. The inter-object relationships of this pattern component describe the interactions among the Node, History, and Viewer Objects. Let $BC_{NavObs} = \langle P, IP, OP, IM, OM, IM_I, OM_I, A\rangle$ be the behavioral contract of Navigational Contexts pattern component and its CCS processes be CCS(BC_{NavObs}), where:

- the set of processes in the design component is $P = \{Node, History, Viewer\}$;
- the set of input ports is $IP = \{N2H, H2V, H2N, V2H, Self, Input\}$;
- the set of output ports is $OP = \{N2H, H2V, H2N, V2H, Self\}$;
- the set of input messages is $IM = \{Navigate, Activate, Record, Backtrack, GoTo, Update\}$;
- the set of output messages is $OM = \{Activate, Record, Update, Backtrack\}$;
- the set of input messages in the interface is $IM_I = \{Navigate, GoTo\}$;
- the set of output messages in the interface is $OM_I = \{\}$;
- the set of actions is $A = \{Navigate, Activate, Record, Backtrack, GoTo, Update\}$;
- the CCS processes are defined by CCS(BC_{NavObs}), where

```
Node(Name, N2H, H2N) ::=
        in(Input, Navigate).out(Self, Activate).out(N2H, Record).action(Navigate).Node(Name, N2H, H2N)
      + in(Self, Activate).action(Activate).Node(Name, N2H, H2N)
      + in(H2N, Activate).action(Activate).Node(Name, N2H, H2N)
History(Name, N2H, H2V, V2H, H2N) ::=
        in(N2H, Record).out(H2V, Update).action(Record).History(Name, N2H, H2V, V2H, H2N)
      + in(V2H, Backtrack).out(H2N, Activate).action(Backtrack).History(Name, N2H, H2V, V2H, H2N)
      + in(Input, Reset).action(Reset).History(Name, N2H, H2V, V2H, H2N)
Viewer(Name, V2H, H2V) ::= in(Input, GoTo)).out(V2H, Backtrack)).action(GoTo).Viewer(Name, V2H, H2V)
      + in(H2V, Update).action(Update).Viewer(Name, V2H, H2V)
NavigationalObserver ::= Node(aNode, N2H, H2N)
     | History(aHistory, N2H, H2V, V2H, H2N)
     | Viewer(aViewer, V2H, H2V)
```

4.2.3. *Active Reference pattern*

In many hypermedia applications, particularly those with spatial or time structures, the user usually needs to have visual knowledge about the current location in terms of space or time during the navigation. A good solution is to maintain an active and perceivable navigational object acting as an index for other navigational objects. In this way, the user is able to interact with both the index and the target nodes. In particular, the user can choose to go to another target node by clicking on the index

Copyright © 2005 John Wiley & Sons, Ltd.

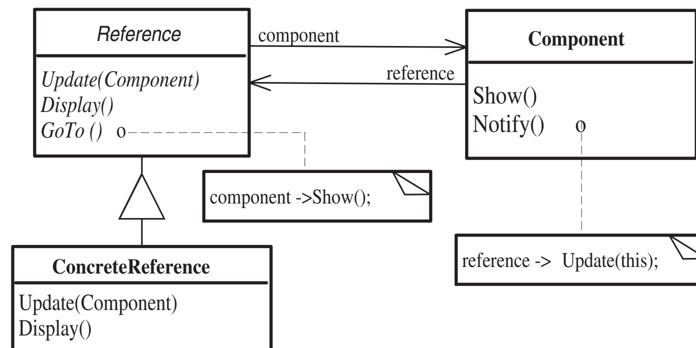

Figure 6. The Active Reference pattern (class diagram).

that is permanently displayed on the screen. The advantage of having an active reference is that the user may have a sense of where they are in the hyperspace at any time. Providing this information not only helps the user to determine the current position in the complex navigation space, but also allows them to change to other positions. The Active Reference pattern [21] was proposed to address this issue by providing a perceivable and permanent reference about the current status of navigation. The current status is usually highlighted. One common example of the application of the Active Reference pattern is to keep an index permanently visible on the screen while navigating a multi-page document. A UML description of this pattern is shown in Figure 6. 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.

The UML collaboration diagram shown in Figure 7 describes the dynamic 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 Figure 7, represents a communication between two objects. The messages `A1:r.GoTo` and `A2:c.Show` present the 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 the second sequence of operations in a similar way.

Copyright © 2005 John Wiley & Sons, Ltd.

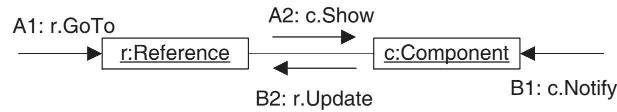

Figure 7. The Active Reference pattern (collaboration diagram).

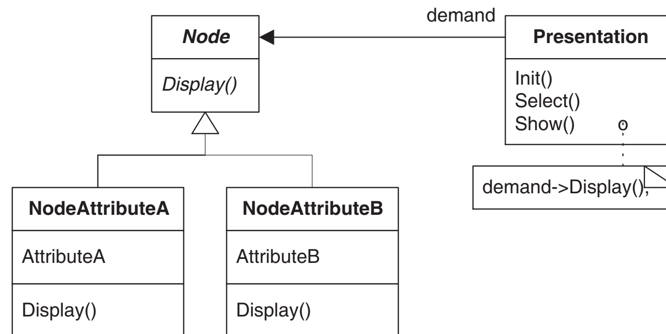

Figure 8. The Information on Demand pattern (class diagram).

4.2.4. Information on Demand pattern

The intent of this pattern is to organize and prioritize large amounts of hypermedia information that cannot be displayed in one screen and let the user decide which items to examine further. It is usually difficult to decide how to show the attributes and anchors of a node in an aesthetically and cognitively pleasing manner, especially when there is large amount of information to be displayed on a small screen. There is usually no other media that can be used for technological or cognitive reasons. The solution to this problem is to select the most important attributes and anchors of a node to be presented in the interface and provide a set of active interface objects, such as buttons, to represent the rest of the attributes and anchors of the node, which can be displayed on demand. For example, a painting may have many attributes, but not all of them can (or need to) be shown at the same time. Hence, a selective set of attributes can be displayed at first. Other attributes can be presented when needed.

Information is often not isolated from its presentation. If the presentation of pages needs to change, that information has to be changed on a file-by-file basis. This pattern isolates the information related to presentation so that the user can change the presentation easily.

The class diagram for the structural aspect of this pattern is shown in Figure 8. The *Node* class and its descendants represent a hypermedia component with different attributes. Each attribute is a significant aspect of a hypermedia component. For example, a painting can have many attributes, such as its graphical presentation attribute, its author attribute, and its textual information attribute. Only a selected

Copyright © 2005 John Wiley & Sons, Ltd.

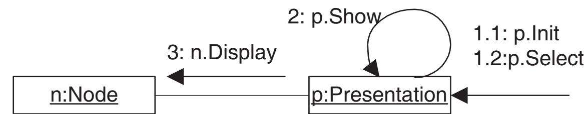

Figure 9. The Information on Demand pattern (collaboration diagram).

number of attributes are displayed on the screen at any point in time. The *Node* class is an abstract class that defines the interface of a hypermedia component. Every descendant of the *Node* class, such as *NodeAttributeA* and *NodeAttributeB*, contains an attribute and provides a *Display* operation to show the attribute on demand. The *Presentation* class provides some mechanisms to initialize the presentation of a hypermedia component and to show additional information on demand. Initially, some default attributes may be displayed and all the rest of attributes are contracted to a set of buttons. This process can be achieved by invoking the *Init* operation in the *Presentation* class. At a latter time, the user can select some attributes to examine, that are not initially displayed, by invoking the *Select* operation and the *Show* operation in the *Presentation* class.

The collaboration diagram is shown in Figure 9. When the user wants some information that is not on display, they can first select the related attributes and then choose to show them. This action will send a message *n.Display* to the Node object to display the information.

4.3. Integration

In the previous section, we have discussed how to model hypermedia design patterns as design component contracts formally and how to apply these design components to address the hypermedia design issues described in Section 4.1. To construct a software design that fulfills all the requirements, we need to compose these design component contracts. The integration operations have been defined in Section 3 and Appendix A.

There are some issues to be considered during the integration phase. First, some classes and objects may play different roles when they are the overlapping parts of two different design component contracts. We have to make sure that the generic elements in these overlapping parts are instantiated with the same names. Second, the integration of two or more design components may cause undesired interactions among them. Some properties or constraints of a design component may not hold after integration. In this case, integration is not allowed because of these constraint violations. The analysis of these interactions is discussed in the next section.

Figure 10 shows one possible integration, 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 represent a pattern instance explicitly 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 [26]. Each class in Figure 10 is attached with a tag containing the corresponding pattern(s) in which it participates and the corresponding role(s) it plays in the pattern(s) in the form of ‘pattern:role’. Symmetrically, each operation (or attribute)

Copyright © 2005 John Wiley & Sons, Ltd.

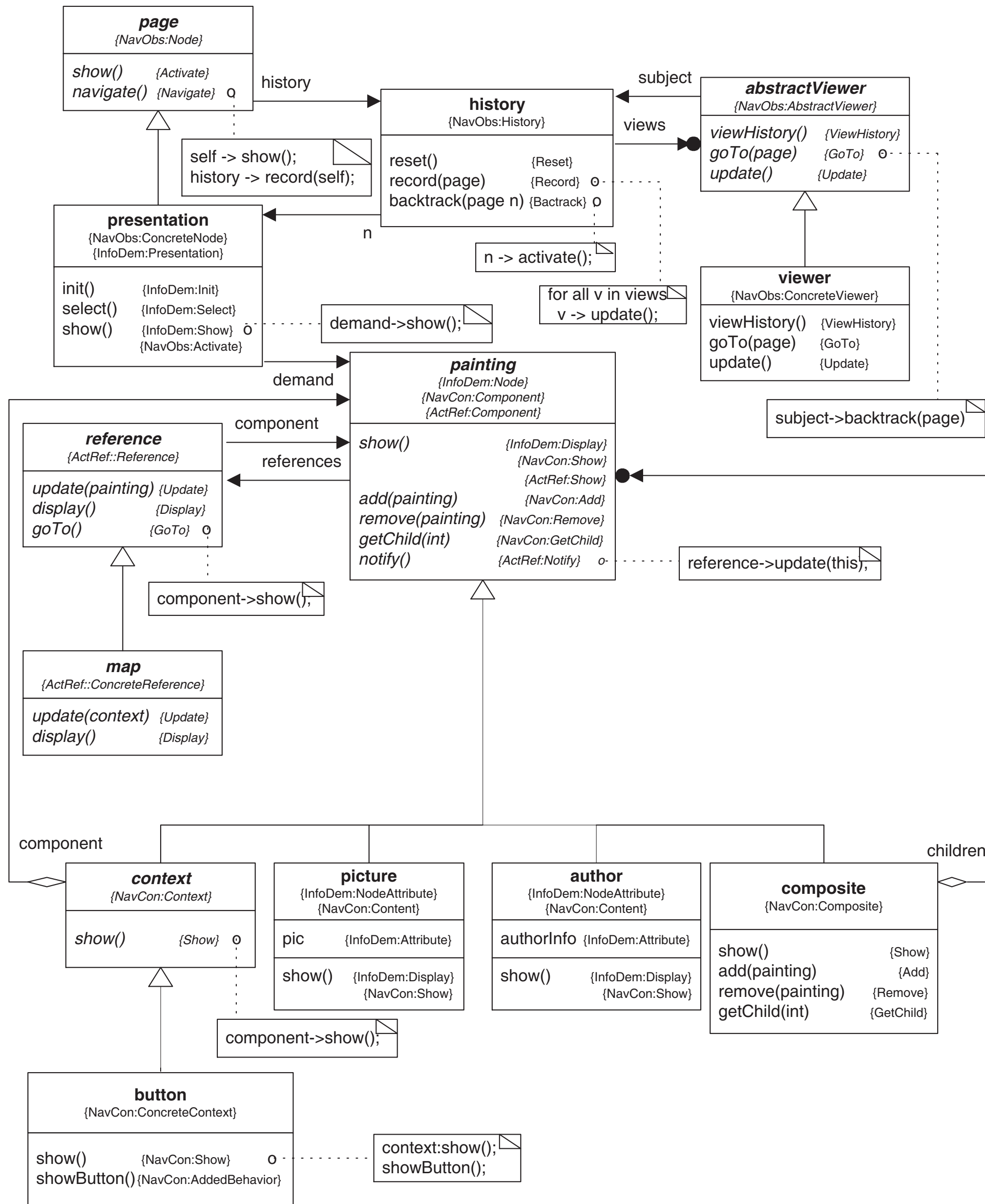

Figure 10. Version 1 of design composition (class diagram).

Copyright © 2005 John Wiley & Sons, Ltd.

is attached with a tag containing the corresponding pattern(s) in which it participates and the corresponding role(s) it plays in the pattern(s). For instance, the *painting* class participates in two patterns: the Active Reference pattern and the Navigational Contexts pattern. It plays the role of *Component* in both patterns. The Active Reference, Navigational Contexts, Information on Demand and Navigational Observer patterns are abbreviated as 'ActRef', 'NavCon', 'InfoDem', and 'NavObs', respectively. For brevity, only the participant name is shown if there is no ambiguity, For example, the *composite* class participates as the *Composite* in the Navigational Contexts pattern.

Example 4.5. Consider the structural instances of the Active Reference, Navigational Contexts, Navigational Observer and Information on Demand components previously defined as SC_{ActRef}, SC_{NavCon}, SC_{NavObs} and $SC_{InfoDem}$, respectively. Let the integration of these structural contracts be $SC_{Combine} = \langle C, AV, M, T, AR, PS\rangle$, where:

- 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 that specify a structural aspect of a design component is $PS = RP \cup CP \cup AP \cup SP \cup QP$, where

RP = {
abstractclass(Painting),
class(Picture),
class(Author),
class(Presentation),
abstractclass(Page),
class(ConcretePage),
class(History),
abstractclass(AbstractViewer),
class(Viewer),
class(Painting),
class(Map),
abstractclass(Reference),
abstractclass(Component),
abstractclass(Context),
class(Button),
class(Content),
class(Composite),
variable(Painting,private,reference,Reference),
variable(Presentation, private, Demand, Node),
variable(Picture, private, Attribute),
variable(Author, private, Attribute),
variable(Page, private, Histories, History),
variable(AbstractViewer, private, Subject, History),
variable(History, private, Views, AbstractViewer),
variable(Context, private, Components, Component),
variable(Composite, private, Children, Component),
method(Presentation, public, Init),
method(Presentation, public, Select),
method(Presentation, public, Show)
method(Picture, public, Display)
method(Author, public, Display)
method(Page, public, Activate),
method(Page, public, Navigate),
method(ConcretePage, public, Activate)
method(History, public, Reset),
method(History, public, Record),
method(History, public, Backtrack),
method(AbstractViewer, public, ViewHistory),
method(AbstractViewer, public, GoTo, Page),
method(AbstractViewer, public, Update)
method(Viewer, public, ViewHistory),
method(Viewer, public, GoTo),
method(Viewer, public, Update)

Copyright © 2005 John Wiley & Sons, Ltd.

method(Painting, public, Show),
method(Painting, public, Notify),
method(Reference, public, Update, void),
method(Reference, public, Display),
method(Reference, public, GoTo)
method(Map, public, Update),
method(Map, public, Display)
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),
method(Button, public, Show, void),
method(Button, public, ShowButton, void),
}
CP = {
inherit(Painting, Picture),
inherit(Painting, Author),
inherit(Page, ConcretePage),
inherit(AbstractViewer, Viewer),
inherit(Reference, Map),
inherit(Component, Content),
inherit(Component, Context),
inherit(Component, Composite)
inherit(Context, Button),
}

AP = {
invoke(Page, Navigate, self, Activate),
invoke(Page, Navigate, Histories, Record),
invoke(History, Backtrack, n, Activate),
invoke(AbstractViewer, GoTo, Subject, Backtrack),
invoke(Presentation, Show, Demand, Display),
invoke(Painting, Notify, reference, this),
invoke(Reference, GoTo, Painting, Show),
invoke(Context, Show, Components, Show),
invoke(Button, Show, Context, Show),
invoke(Button, Show, Button, ShowButton)
}
SP = {
member(Picture, NodeAttributeSet),
member(Author, NodeAttributeSet),
member(Map, ConcreteReferenceSet),
member(ConcretePage, ConcreteNodeSet),
member(Viewer, ConcreteViewerSet),
member(V, Views),
member(Button, ConcreteContextSet),
member(Child, Children)
}
QP = {
forall(member(V, Views), CI),
forall(member(Child, Children), CO)
}
CI = {
invoke(History, Record, V, Update)
}
CO = {
invoke(Composite, Show, Child, Show)
}

Example 4.6. 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 $\text{CCS}(BC_{Combine})$, where:

- 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

In the next 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.

5. DESIGN ANALYSIS

Each design component possesses some properties before being composed with other design components. These properties characterize this component and should hold at any time. Without these properties, the component may have changed its identity. The composition of design components may introduce inconsistencies or undesired interactions among the design components for several reasons. First, the classes or objects that share the same name in a composition may play different roles in different components. We need to make sure these roles are consistent. Second, the integration of two or more design components may cause each component to gain or lose some of its properties. Some properties or constraints may not hold after the integration. In addition, new properties that belong to none of the components may be introduced in the composition. Although these new properties may not have any conflicts with the properties of each component, they may be unexpected by the user of the composition. In these cases, the user may need to change the way components are composed, or replace some components with new ones.

After the design component is composed with other components, we need to ensure that the desired properties of this component still hold in the composition. The choice of properties we analyze are based on our experience and preliminary investigation [5,27] of the design component space. These properties are normally specified in an application-independent manner. When a design component contract is instantiated in an application, the properties corresponding to the design component can be instantiated using the application-domain information. These instantiated properties are then checked when the instance of the design component contract is composed with others. The representation of these properties can be re-instantiated and reused in another application when the corresponding design component contract is applied in that application.

Composition errors can be difficult to detect by visual inspection of the design diagram. If these errors are left undetected and are propagated into implementation errors, they become even harder to detect because they are transformed and blended into complex implementation structures. Furthermore, manually finding errors in software design compositions is tedious and error-prone. The goal of our

Copyright © 2005 John Wiley & Sons, Ltd.

design analysis is to be able to find design composition errors, with the help of automated verification tools.

In [15], we have classified various properties that can be verified within the design component contracts. Our goal is to find if there are any inconsistencies or errors in the composition of the design components. In the remainder of this section, we show how Prolog and XMC are used to check different properties against the integration of design component contracts described previously, to identify certain faults, and to verify that the modifications of these designs eliminate these faults. In particular, we show the analysis of a number of structural properties (consistency and integrity properties), behavioral properties (global, concurrency, sequence and occurrence properties), evolutionary properties, and hybrid properties in the following sections.

5.1. Structural analysis

As discussed previously, the structural aspect of a design component contract is implemented in XSB Prolog. The structural properties are also implemented in XSB Prolog. Thus, in this case, the property specification language is the same as the model specification language. A structural property is represented as a question in XSB Prolog. When a question is asked, Prolog will search through the knowledge base containing the facts about the structure model. It unifies the facts that match the fact in the question. Two facts are matched if their predicates are the same (spelled the same way), and if their corresponding arguments are the same. If a question can be inferred from the facts currently in the knowledge base, Prolog will respond 'yes'. That is, the fact is a property of the component structure. If no such fact exists in the knowledge base, Prolog will respond 'no'. That means the fact is not a property of the component structure. A Prolog question can also have variables[¶] that indicate universal quantification as its arguments. When a question has a variable argument, the variable can be either instantiated or not instantiated. A variable is instantiated when there is a corresponding object. On the other hand, a variable is not instantiated when the corresponding object is not yet known. When Prolog is asked a question containing a variable, Prolog searches through all of its facts to unify an object that corresponds to the variable. A variable does not name a particular object in itself, but it can be used to stand for objects that we cannot name. Using variables, we can specify properties that are more complex. In this way, we can check properties described in clausal form logic against the specification of the composition model of structural contracts.

A structural property is an expression of the form

$$H \leftarrow B_1, \ldots, B_n, \quad n \geq 0 \tag{1}$$

where $H, B_1, \ldots, B_n$ are atoms. H is the head of the property, whereas, $B_1, \ldots, B_n$ are the body of the property. We classified the properties according to the predicate symbols they contain.

For example, the question whether there are multiple-inheritance relationships in an object-oriented design can be represented by

$$\texttt{multiple_inherit} :- \texttt{inherit(A, B)}, \texttt{inherit(C, B)}, \texttt{A} \neq \texttt{C}.$$

[¶]A variable is a term beginning with a capital letter in Prolog. A variable can be matched to the terms in the knowledge base by unification.

where `inherit(A, B)` represents that `B` is a subclass of `A`. When we submit this question to Prolog's inference engine, Prolog will reply 'yes' if a multiple-inheritance relationship is found, or 'no' if not. We note that there is no argument for `multiple_inherit` in this case since we only want to know whether there are multiple-inheritance relationships in this design. If we wanted to know whether a specific class 'b' has multiple parents, we could have used `multiple_inherit` having the name of the class as an argument as follow: `multiple_inherit(A, b, C) :- inherit(A, b), inherit(C, b) A ≠ C.`

5.1.1. Consistency properties

A consistency property refers to a situation where two facts are in conflict and cannot both be true at the same time. As defined in [15], a consistency property is a rule property that includes the following.

- CP1. $H \leftarrow \mathit{class}(X), \mathit{abstractclass}(X)$. If H is true, then X is defined as both concrete class and abstract class.
- CP2. $H \leftarrow \mathit{variable}(A, _, C, _), \mathit{method}(A, _, C, _)$. If H is true, then C is defined as both a variable and a method in a class A.
- CP3. $H \leftarrow \mathit{inherit}(A, A)$. A class cannot inherit from itself.
- CP4. $H \leftarrow \mathit{inherit}(A, B), \mathit{inherit}(B, A)$. Two classes cannot inherit from each other.

We have described property implementation in [15].

Example 5.1. Consider a Prolog implementation of the first consistency property (CP1)

$$\mathtt{consistency1(X) :- class(X), abstractclass(X).}$$

which is checked against the first version of the formal integration contract $SC_{Combine}$ described Example 4.5. The UML diagram of this integration is shown in Figure 10. The verification result is

```
| ?- consistency1(X).

X = painting;

no
| ?-
```

The verification result showed that there was a 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 in that the Active Reference pattern defines the *Component* class, whose instance is the *painting* class, as a concrete class whereas the Navigational Contexts pattern defines the *Component* class, whose instance is the *painting* class, as an abstract class. This means that the *Component* class in the Active Reference pattern cannot be mapped to the *painting* class that has the role of *Component* in the Navigational Contexts pattern and the role of *Node* in the Information

Copyright © 2005 John Wiley & Sons, Ltd.

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 in the Active Reference pattern should have association relationships with the *picture* and *author* classes instead of with the *painting* class in Figure 10. The modified design is shown in Figure 11.

5.1.2. Integrity properties

An integrity property refers to a situation where one fact cannot be true independent of other facts. Integrity properties include properties for roles and properties for actions. A role integrity property ensures that a variable or a method cannot be defined without definitions of their classes. An action integrity property ensures that the action predicates are performed on a well-defined basis.

In the following, we demonstrate the verification of one role integrity property and one action integrity property.

Example 5.2 (Role integrity property). Consider the property that if a state variable is defined in a class then the class should be defined as well. Therefore, there should not have be defined state variable without its class being defined as either an abstract class or a concrete class. Formally, it is defined by

$$role1(A) \leftarrow variable(A, _, C, _), not(abstractclass(A)), not(class(A)).$$

In this case study, the property can be implemented by

```
role1(A) :- variable(A, _, C, _), not(abstractclass(A)), not(class(A)).
```

The verification result is shown as follows:

```
| ?- role1(A).

no
| ?-
```

The result shows that *role1(A)* does not hold, which confirms that the class is defined for each defined state variable.

Example 5.3 (Action integrity property). Consider the property when a method B, defined in a class A, invokes another method D, defined in class C, both classes A, C and methods B, D should be defined. Thus, there is something wrong if either classes A, C or methods B, D are not defined. Formally, it is defined by

$$\begin{aligned} &action1(A, B, C, D) \leftarrow invoke(A, B, C, D),\\ &\quad \{not(class(A)); not(class(C)); not(method(A, _, B, _)); not(method(C, _, D, _))\}. \end{aligned}$$

In this case study, the property can be implemented by:

```
action1(A, B, C, D) :- invoke(A, B, C, D), {not(class(A));
                 not(class(C)); not(method(A, _, B, _));
                 not(method(C, _, D, _))}.
```

The verification result is shown as follows:

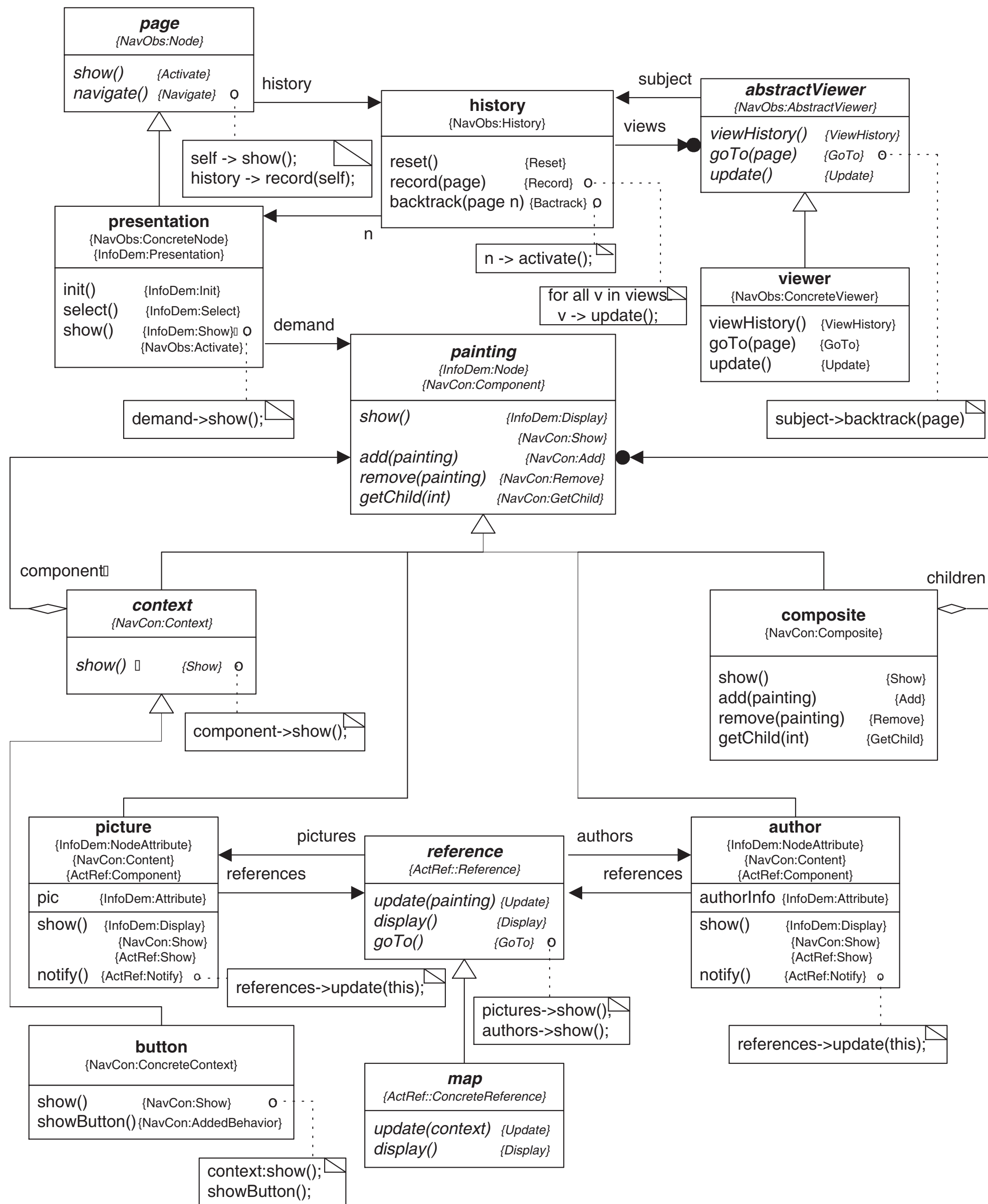

Figure 11. Version 2 of design composition (class diagram).

Copyright © 2005 John Wiley & Sons, Ltd.

```
| ?- action1(A, B, C, D).

no
| ?-
```

The result shows that *action1*(A, B, C, D) does not hold since its classes and methods are not defined for each invocation.

5.2. Behavioral analysis

A behavioral property ϕ is defined using μ-calculus. The semantics of the modal μ-calculus can be described over sets of states of labeled transition systems (LTSs). Temporal properties can be defined as formulae recursively using the fixed point operators, where μ is the least-fixed point operator and ν is the greatest-fixed point operator. The least-fixed point operator represents the computation starting from the minimal element and then iteratively expanding whereas the greatest-fixed point operator starts from the maximal element and iteratively reducing. In general, ν formulae imply safety properties that require that a result never occurs, whereas μ formulae refer to liveness properties that require that a result must occur. Based on the classification of basic properties into safety and liveness [28], it is reasonable to say that ν is safety and μ is liveness [29].

The syntax of μ-calculus is

$$\Phi ::= Z \parallel tt \parallel ff \parallel \Phi \wedge \Phi \parallel \Phi \vee \Phi \parallel \langle A \rangle \Phi \parallel [A]\Phi \parallel \mu Z.\Phi \parallel \nu Z.\Phi$$

where Z is a set of formula variables, A is a set of actions, tt and ff are propositional constants, $\wedge$ and $\vee$ are standard logical connectives, $\langle A \rangle \Phi$ denotes that possibly after the action A the formula Φ holds, $[A]\Phi$ denotes that necessarily after the action A the formula Φ holds[‖]; that is, formula ϕ holds in all reachable states (within one step) by an action a transition. The formula $\mu Z.\Phi$ and $\nu Z.\Phi$ stand for least-fixed point and greatest-fixed point, respectively.

Alternation depth of a μ-calculus formula refers to the level of non-trivial nesting of fixed points within it where adjacent fixed points are of a different type. In other words, it can be loosely defined as the maximum number of μ/ν alternations in a chain of nested fixed points. Formulae with alternation depth greater than two are difficult to understand [30]. All CTL [31] formula can be expressed in the μ-calculus with alternation depth 1, whereas properties associated with fairness require alternation depth 2 [32]. Moreover, all modal or temporal logics of programs (e.g. LTL, CTL, ACTL, CTL*) can be translated into μ-calculus with alternation depth 1 or 2 [33]. For example, the CTL assertion $AG\Phi$ denotes that for all computation paths Φ is always true. This can be characterized by μ-calculus as $\nu Z.\Phi \wedge [-]Z$ that is the greatest-fixed point of functional $\Phi \wedge [-]Z$. Z is an atomic proposition variable ranging over sets of states. $[-]$ ($\langle - \rangle$) can be translated to AX (EX) in CTL. Thus, $\Phi \wedge [-]Z$ means that Φ holds in the current state and holds in the next states of all paths. Other CTL formula, such as $EF\Phi$, $AF\Phi$ and $EG\Phi$, can be characterized by μ-calculus as $\mu Z.\Phi \vee \langle - \rangle Z$, $\mu Z.\Phi \vee [-]Z$ and $\nu Z.\Phi \wedge \langle - \rangle Z$, respectively.

While XL can be used to implement the processes of a behavioral contract, it is not powerful enough to directly specify interesting temporal relationships between system actions. Since such temporal

[‖] '-' represents any action in ⟨-⟩ or [-]. It also represents complement in ⟨-A⟩ or [-A].

Copyright © 2005 John Wiley & Sons, Ltd.

properties are often required in concurrent systems or reactive systems, model checkers usually support a temporal property specification language, such as one based on μ-calculus for XMC model checker or based on LTL or CTL. The behavioral properties are specified in μ-calculus (the property specification language of XMC) instead of XL (the model specification language of XMC).

5.2.1. Global properties

We have defined two global properties: *invariant* and *never*. An *invariant* property describes that something is always true, whereas a *never* property depicts that something never happens. We demonstrate the verification of an *invariant* property in this case study. The *never* properties can be verified similarly [15].

Example 5.4 (Invariant). Consider the behavioral contract $BC_{Combine}$ of the integration described in Example 4.6. According to the Information on Demand pattern discussed in Section 4.2.4, the state variable 'pic' in the 'picture' class (see Figure 11) should always have a value (non-empty) about the corresponding drawing so that there is always a 'picture' on display when a user chooses to 'show' it. Therefore, the invariable property that 'pic' is always true, which is denoted by '*AG Pic*' in CTL, is

$$\nu X.Pic \wedge [-]X$$

The implementation of these properties in XMC is shown as**:

```
invariant -= Pic /\ [-] invariant.
```

A test Prolog program is written as follows††:

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

test :-
     write('Invariant '), checkit(mck(combination,invariant)).
```

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

```
| ?- test.
Invariant      mck(combination,invariant) is true.

yes
```

where `mck` checks the properties `invariant` against the model `combination`. The results show that the invariant property holds in the integration of the design components.

**XMC uses '$+=$' and '$-=$' to represent least- and greatest-fixed points, respectively.
††`xlc(spec)` is used to load and compile the XL specifications of a model in XMC. `mck(process, formula)` is used to check the property formula against the process model.

5.2.2. *Concurrency properties*

We have defined two kinds of concurrency properties: deadlock and mutual-exclusion. A deadlock property states that a deadlock system state is reachable. When the system gets into this state, no action can proceed. A mutual-exclusion property states that when several processes access a resource, only one process may act on the resource at any given time. The analysis of these concurrency properties is useful in many applications. For example, system deadlock may lead to denial of service in E-commerce systems [34]. Thus, ensuring a system is deadlock-free can help to prevent some kind of attacks. We shall demonstrate the verification of a deadlock property in this case study. The mutual-exclusion properties can be verified similarly [15].

Example 5.5 (Deadlock). Consider the behavioral contract $BC_{Combine}$ of the integration described in Example 4.6. The absence of deadlock can be denoted by

$$\nu X.[-]X \wedge \langle - \rangle tt$$

The implementation of these properties in XMC is shown as follows:

```
deadlockFree -= [-]deadlockFree /\ <->tt.
```

A test Prolog program is written as follows:

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

test :-
      write('Free from deadlock'),
      checkit(mck(combination,deadlockFree)).
```

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

```
| ?- test.
Free from deadlock      mck(combination,deadlockFree) is true.

yes
```

where `mck` checks the properties `deadlockFree` against the model `combination`. The results show that the integration of the design components is free from deadlock.

5.2.3. *Sequence properties*

In [15], we have defined six types of sequence properties: *while, eventual, possible, until, unless* and *then*. In this section, we demonstrate the verification of two *then* properties related to this case study. Other sequence properties can be verified similarly.

Example 5.6 (Then). Consider the behavioral contract $BC_{Combine}$ of the integration described in Example 4.6. 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

Copyright © 2005 John Wiley & Sons, Ltd.

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.

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 (`sequence1` and `sequence2`) are implemented generically in XMC as shown in Appendix A:

A test Prolog program is written as follows:

```
:- 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 follows:

```
| ?- 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. 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 sharing part further down to the concrete context class (the *button* class) as shown in Figure 12. The model-checking results show that both properties hold at this time.

5.2.4. *Occurrence properties*

We have defined two kinds of occurrence properties: even-length path and fairness in [15]. An even-length path property states that there exists a maximal path of even length for an action. A fairness property describes that an action occurs infinitely often if the action is enabled infinitely often. We demonstrate the verification of an even-length path property in this case study.

Copyright © 2005 John Wiley & Sons, Ltd.

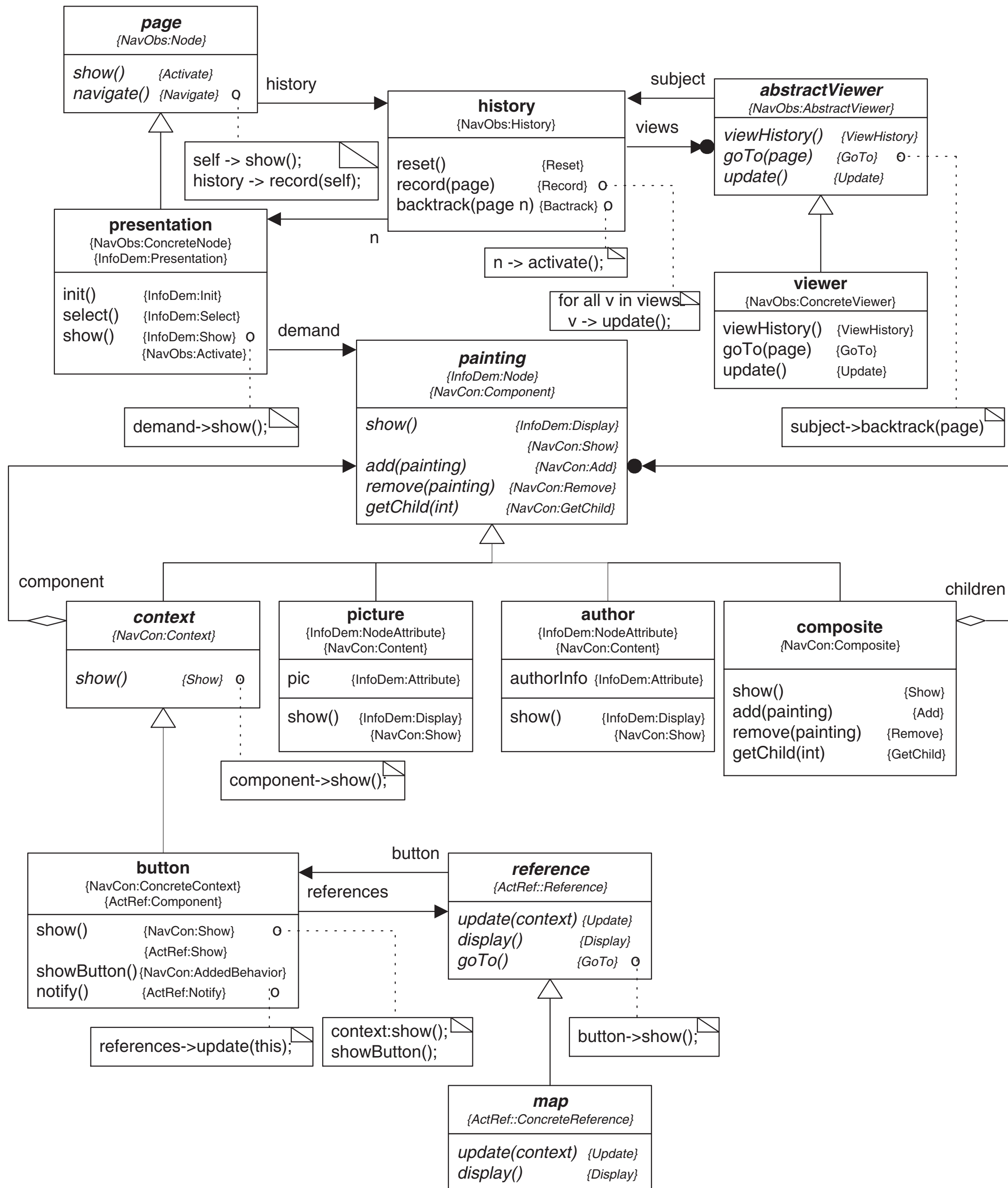

Figure 12. Version 3 of design composition (class diagram).

Copyright © 2005 John Wiley & Sons, Ltd.

Example 5.7 (Even-length path). Consider the behavioral contract $BC_{Combine}$ of the integration described in Example 4.6. The system records the history information of navigation. The history can be reset by a user. To avoid unintentional reset of history, the reset request should be sent twice in order for the history being reset. Since the history may be reset any number of times, there should be an even number of the `reset` actions in the action sequence. The property that there exists an even-length of action `reset` can be denoted by

$$\mu X.[reset]ff \vee \langle reset\rangle\langle reset\rangle X$$

The implementation of these properties in XMC is shown as follows:

```
evenlength += [reset]ff \/ <reset> <reset> evenlength.
```

A test Prolog program is written as follows:

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

test :-
      write('Even length of resets'),
      checkit(mck(combination,evenlength)).
```

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

```
| ?- test.
Even length of resets      mck(combination,evenlength) is false.

yes
```

The results show that it is not true that the reset operations always come in even numbers. The reason for this error is that the original specification of the behavioral contract of the Navigational Observer pattern in Example 4.4 does not require double requests for a history reset. This specification mistake can be corrected by modifying the `History` process of the behavioral contract as follows:

```
History(Name, N2H, H2V, V2H, H2N) ::=
     in(N2H, Record).out(H2V, Update).action(Record).History(Name, N2H, H2V, V2H, H2N)
   + in(V2H, Backtrack).out(H2N, Activate).action(Backtrack).History(Name, N2H, H2V, V2H, H2N)
   + in(Input, Reset).action(Reset).History(Name, N2H, H2V, V2H, H2N)
```

is changed to

```
History(Name, N2H, H2V, V2H, H2N) ::=
     in(N2H, Record).out(H2V, Update).action(Record).History(Name, N2H, H2V, V2H, H2N)
   + in(V2H, Backtrack).out(H2N, Activate).action(Backtrack).History(Name, N2H, H2V, V2H, H2N)
   + in(Input, Reset).action(Reset).in(Input, Reset).action(Reset).History(Name, N2H, H2V, V2H, H2N)
```

By running the test program again, the result shows that the property holds this time:

```
| ?- test.
Even length of resets      mck(combination,evenlength) is true.

yes
```

Copyright © 2005 John Wiley & Sons, Ltd.

5.3. Evolutionary analysis

One of the advantages of using design patterns is that they cope with the evolution of designs. We have defined the *extension* and *retraction* operations for the evolution of the structural contract and behavioral contract of a design component in [15]. A design component can evolve in a restricted and predefined manner by applying these operations. We implement this evolution information in the descriptions of each design component by the `extend_` and `retract_` rules.

When the application such as that in this case study requires another kind of context information, such as *text* information in addition to *button* information, this design decision related to structural change can be implemented by instantiating a predefined Prolog rule as follows: `extend_contexts(context, text, showText, show)`. For example, Van Gogh's painting *SunFlowers* can be reached while exploring paintings about nature through a context link of buttons, it can also contain the information about the natural aspect of the painting *SunFlowers* as another kind of context information. On the other hand, *SunFlowers* can be accessed as a work of Van Gogh. In addition to a context link of buttons connecting Van Gogh's work, the new context (*text*) can contain the information about the relationships with other paintings by Van Gogh. The modified design is shown in Figure 13. It contains one additional concrete context class (*text*). If Van Gogh's painting *SunFlowers* is a value of the attribute *pic* in the class *picture*, it can be attached with two kinds of context information: one is a list of buttons that connect the painting *SunFlowers* with other related paintings; the other is additional information about the painting *SunFlowers* related to the particular context.

After the evolution of design components, we need to ensure that the properties that held before still hold after the evolution. Therefore, we need to check those properties related to the modified design components. In this case study, we check the `sequence2` property described previously. Other properties are checked similarly.

Example 5.8. Consider the behavioral contract $BC_{Combine}$ of the integration described in Example 4.6 and its evolution described previously. The property, which the *show* operation in this new concrete context class will be eventually invoked when the *goTo* operation in the *reference* class is called, is described by instantiating the second sequence property (see Example 5.6) as

`sequence2(reference, goTo, text, show).`

A test Prolog program is written as follows:

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

test :-
      extend_contexts(context, text, showText, show),
      write('Sequence2'),
      checkit(mck(combination, sequence1(reference, goTo, text, show))).
```

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

```
| ?- test.
Sequence2     mck(combination,sequence2(reference, goTo, text, show)) is true.

yes
```

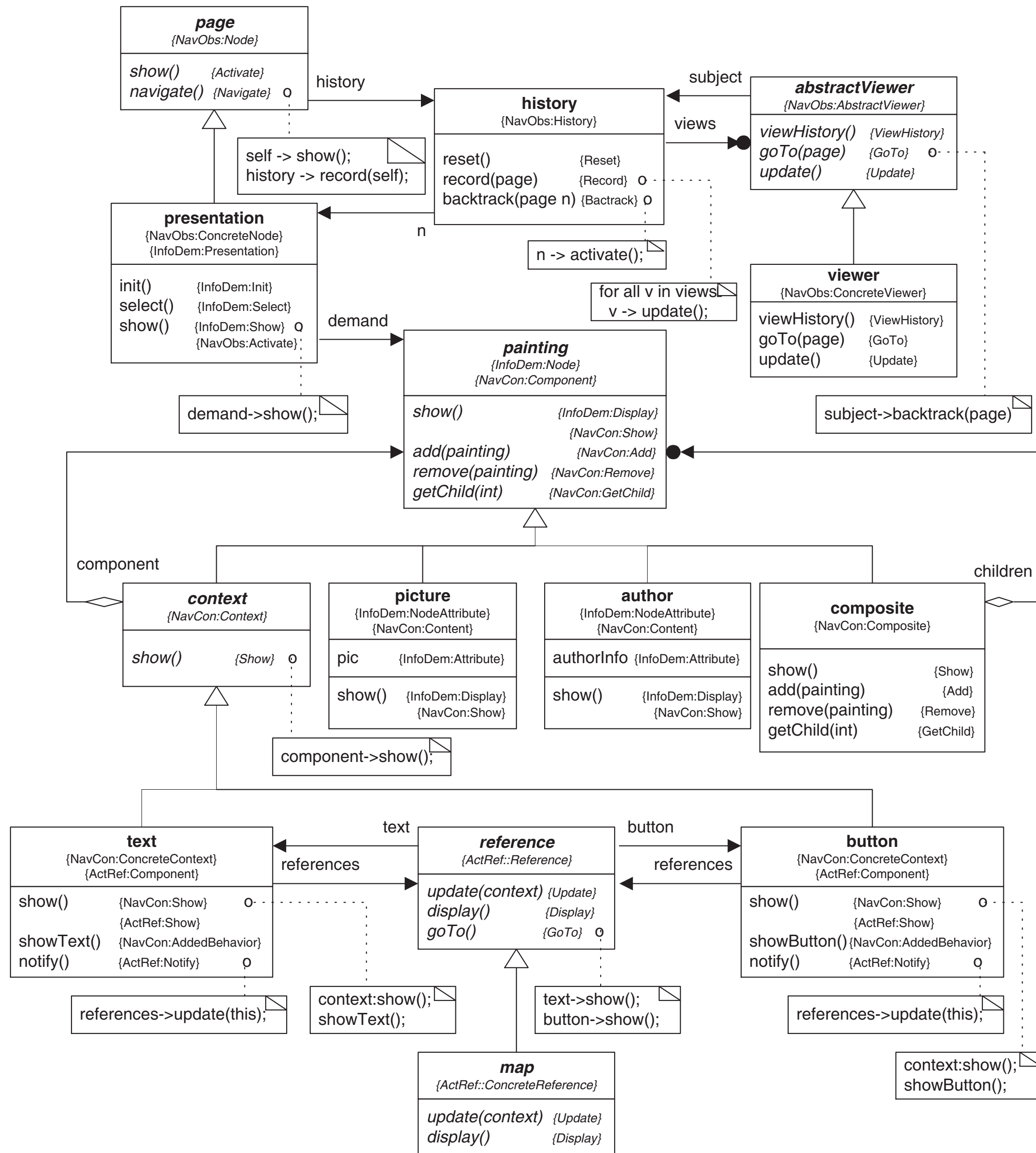

Figure 13. Version 4 of design composition (class diagram).

Copyright © 2005 John Wiley & Sons, Ltd.

As another example, the Navigational Observer pattern separates the history and the viewers of the history so that they can vary independently. In the fourth version of the composed design (see Figure 13), there is one viewer of the history. The history is considered as a simple stack recording each visit to a hypermedia component. Sometimes, it could be necessary to provide viewers with a condensed history, when the same object is visited more than once. Another viewer of the history can be added, which eliminates redundant visits to the history. The modified design with the addition of the condensed viewer is the second version shown in Figure 14. As in the previous example, this evolutionary step requires that the properties related to the modified design components be checked. We omit the checking here.

5.4. Hybrid analysis

In [15], we have defined properties related to a combination of structural and behavioral properties. Although the structural and behavioral properties are based on different formalisms, the verification tools (XSB Prolog and XMC) allow us to check different kinds of properties jointly. Since XMC is written in XSB Prolog, model checking is actually conducted using logic programming by traversing the state space using XSB Prolog. For example, if there is an action A that causes a state transition from state X to state Y, then it is defined as `:- transit(X, A, Y)` in Prolog. In the following, we show one example of a hybrid property.

Example 5.9. (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 Examples 4.5 and 4.6, 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 follows:

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

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 follows:

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

test :-
```

Copyright © 2005 John Wiley & Sons, Ltd.

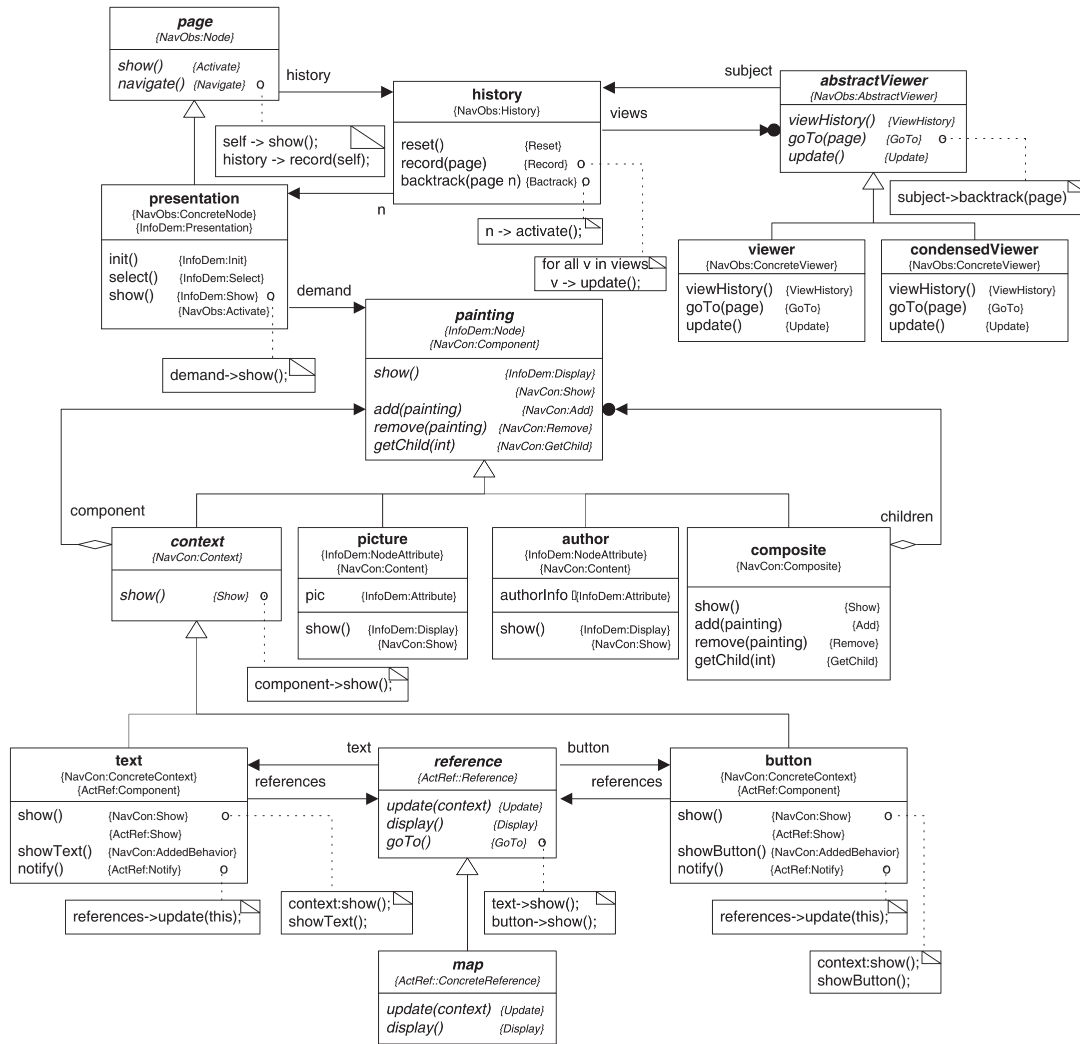

Figure 14. Version 5 of design composition (class diagram).

Copyright © 2005 John Wiley & Sons, Ltd.

```
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 (*checkit*/1), this test program combines the checking of a structural property and a behavioral property in Prolog. By running this Prolog program, the results are shown as follows:

```
| ?- 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.

6. RELATED WORK

The notion of contracts in software development is attributed to Meyer [35]. Another contribution, the object-oriented contracts of Helm *et al.* [36] focused 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 [37], Beugnard *et al.* proposed a four-level contract for components to increase trust. In [38], Borgida and Devanbu propose description logics (DLs) as a formal basis for describing components. In this paper, we defined a formal model of design component based on contracts and a rigorous analysis approach to software design composition based on automated verification techniques.

Keller and Schauer [2] 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 a 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.

Formalizing design and architecture patterns has been proposed in [39,40]. Although Mikkonen [40] has discussed the composition of two design patterns based on a formal method, his approach relies on a particular 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. Moreover, Mikkonen's approach focuses on formalizing design patterns, whereas our work deals with a more general approach to design components [2] and their composition.

Model-checking of implicit-invocation systems has been investigated in [41]. 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

Copyright © 2005 John Wiley & Sons, Ltd.

method invocation and the mechanisms that implement event dispatch and event delivery policies. This abstraction and modeling process is highly domain-specific. Model checking the Java meta-locking algorithm has also been conducted in [42]. Modeling techniques for one class of software systems may be completely inappropriate for another. Our work on modeling behavioral contracts can be seen as another domain-specific model-checking framework.

Automated extraction of high-level models from Java source code has been proposed in the Bandera project [43]. Properties can be checked against these high-level models using model checkers, such as SMV [44] and SPIN [45]. Although these source-code analysis technologies can be used to analyze prototypes of designs, the types of properties that can be verified and systems that can be modeled are limited to the underlying technologies of the model checkers, mostly, based on LTL and CTL. In addition, the high-level models that are extracted are usually specified using the model specification languages of the model checkers, which do not correspond to understandable design models such as those based on UML. Our goal is different. We construct a generic model for design components, specify each design component, and verify their compositions.

Informal object-oriented modeling techniques based on graphical notations, such as UML [46], are often used to analyze and specify user requirements and software designs. These visual notations, such as boxes and lines, provide specifications that are easy to use and understand. Nevertheless, without a precise semantic basis for the notations used in most informal techniques, the potential for rigorous analysis is limited. Recent development of standard object-oriented modeling notations provides the syntax and semantics of the notations in terms of a meta-model expressed in a combination of natural language descriptions, UML notations and the Object Constraint Language (OCL) [47]. OCL provides for specification of program invariants and pre/post conditions on operations. Weaknesses of OCL-based approaches include the lack of ability to reason about such specifications and the fact that current tool support does not provide a mechanism to transfer from such specifications to code. It has been pointed out that these natural language descriptions are not sufficient to express the semantics of the UML notation precisely [48,49]. In addition, since OCL is used to express complex constraints that the UML notations cannot express, semantic analysis and refinement of any UML model should deal with these constraints.

7. CONCLUSIONS

In this paper, we have introduced an overview of a formal model of design components based on contracts, and a rigorous analysis approach to software design composition based on automated verification techniques. The definition and analysis of design component contracts are based on logic programming. Discovering composition errors at the design level is typically much easier than at the code level because a small design component 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. A Web-based hypermedia application has been used to demonstrate our approach and several system properties have been analyzed.

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

Copyright © 2005 John Wiley & Sons, Ltd.

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.

Our approach has been evaluated through a case study in the hypermedia domain. The goal of this case study is to show the applicability and accuracy of the approach by illustrating the features and the kinds of properties that can be checked. In the course of this experience, we have found the following. First, there are many different ways of composing design components with each approach usually rendering different results. Thus, it is often difficult to know which way may cause inconsistencies by visual inspection of the UML diagrams. Rigorous analysis is helpful for finding the inconsistencies and changing to the correct integration. Second, it is not always true that there are inconsistencies in the design model when a fault is detected by a model checker. It is possible that the property specification is wrong or, at least, does not correctly express the desired requirement. Third, many inconsistencies can be detected and resolved by rearranging the composition. However, some cannot be solved easily by rearranging the composition, which leaves two alternatives. One alternative is to replace the troublesome component with a new component providing the required functionality. Nonetheless, the new component may itself generate new inconsistencies. The other alternative is to live with these inconsistencies since some of them are inherited and may not cause a problem. However, it is important to be aware of their existence and avoid possible issues in an application. Finally, inconsistencies are more likely to happen in the overlapping parts of a composition. In fact, they are more subtle and difficult to detect if they happen outside the overlapping parts of a composition.

APPENDIX A

A.1. Predicate symbols

Predicates belonging to $\mathcal{RP}$, $\mathcal{CP}$, $\mathcal{AP}$, $\mathcal{SP}$ and $\mathcal{QP}$ are listed in the following tables. The atoms and literals that can be specified using the predicate symbols are the corresponding names in each design component.

$\mathcal{RP}$ predicate	Arity	Argument types	Meaning
class	1	*CT*	If class(c) is true and $c \in CT$, then c plays the role as a class in the component.
abstractclass	1	*CT*	If abstractclass(c) is true and $c \in CT$, then c plays the role as an abstract class in the component.
variable	4	*CT*, *ART*, *AVT*, *TT*	If variable(c, ar, a, t) is true and $c \in CT$, $ar \in ART$, $a \in AVT$, and $t \in TT$, respectively, then a is defined in c with type t. It can be accessed with the right ar.
method	4	*CT*, *ART*, *MT*, *TT*	If method(c, ar, m, t) is true and $c \in CT$, $ar \in ART$, $m \in MT$, $t \in TT$, then m is a method defined in class c with return type t. It can be accessed with the right ar.

Copyright © 2005 John Wiley & Sons, Ltd.

$\mathcal{CP}$ predicate	Arity	Argument types	Meaning
inherit	2	*CT*, *CT*	If inherit(a, b) is true and $a, b \in CT$, then b is a subclass of a.
association	2	*CT*, *CT*	If association(a, b) is true and $a, b \in CT$, then a, b are connected with association relation.
aggregation	2	*CT*, *CT*	If aggregation(a, b) is true and $a, b \in CT$, then a, b are connected with aggregation relation.

$\mathcal{AP}$ predicate	Arity	Argument types	Meaning
invoke	4	*CT, MT, CT, MT*	If invoke(c_1, m_1, c_2, m_2) is true and $c_1, c_2 \in CT$, $m_1, m_2 \in MT$, then method m_2, which is defined in class c_2, calls the method m_1, which is defined in c_1.
return	3	*CT, MT, CT*	If return(c, m, o) is true and $c, o \in CT$, $m \in MT$, then the method m, defined in class c, returns an object o of type *CT*.
new	3	*CT, MT, CT*	If new(c, m, o) is true and $c, o, \in CT$, $m \in MT$, then the method m, defined in class c_1, create a new object o of type *CT*.

$\mathcal{SP}$ predicate	Arity	Argument types	Meaning
element	2	$CT, 2^{CT}$	If element(e, s) is true and $e \in CT$, $s \in 2^{CT}$, then e is an element of set s (i.e., $e \in s$).

$\mathcal{QP}$ predicate	Arity	Argument types	Meaning
forall	2	*SPT, PT*	If forall(a, b) is true and $a = element(e, s)$, $b \in PT$, then for all $e \in s$, b is true.
exist	2	*SPT, PT*	If exist(a, b) is true and $a = element(e, s)$, $b \in PT$, then there exist $e \in s$, b is true.

A.2. Behavioral integration

Definition A.1 (Behavioral interfaces). Let $\mathcal{BC} = \langle P, IP, OP, IM, OM, IM_I, OM_I, A\rangle$ be the behavioral contract of a design component. The interface of behavioral contract is a tuple $\mathcal{BI} = \langle IM_I, OM_I, IP_I, OP_I, P_I\rangle$, $\mathcal{BI} \subset \mathcal{BC}$, where IM_I is the set of input messages sent from outside the design component. OM_I is the set of output messages sent outside the design component. IP_I is the set of input ports that are used to receive input messages in the set IM_I. Thus, $IP_I = \{ip \in IP \mid \exists i \in IM_I, imessage(i) = ip\}$. OP_I is the set of output ports that are used to send output messages in the set OM_I. Thus, $OP_I = \{op \in OP \mid \exists o \in OM_I, omessage(o) = op\}$. P_I is the set of processes that receive (send) input (output) messages in the set IM_I (OM_I). Thus, $P_I = \{p \in P \mid \forall i, o, i \in IM_I, o \in OM_I, iport(i) = p \vee oport(o) = p\}$.

Definition A.2 (Behavioral integration). Let $BC^1 = \langle P^1, IP^1, OP^1, IM^1, OM^1, IM_I^1, OM_I^1, A^1\rangle$ be the behavioral contract of design component 1, with interface $BI^1 = \langle IM_I^1, OM_I^1, IP_I^1, OP_I^1, P_I^1\rangle$. Its CCS-process is CCS(BC^1). Let $BC^2 = \langle P^2, IP^2, OP^2, IM^2, OM^2, IM_I^2, OM_I^2, A^2\rangle$ be the behavioral contract of design component 2, with interface $BI^2 = \langle IM_I^2, OM_I^2, IP_I^2, OP_I^2, P_I^2\rangle$. Its CCS-process is

CCS(BC^2). The behavioral composition is defined by $BC = \langle P, IP, OP, IM, OM, IM_I, OM_I, A\rangle$, with interface $BI = \langle IM_I, OM_I, IP_I, OP_I, P_I\rangle$. Its CCS-process is CCS(BC). $P = P^1 \cup P^2$; $IP = IP^1 \cup IP^2$; $OP = OP^1 \cup OP^2$; $IM = IM^1 \cup IM^2$; $OM = OM^1 \cup OM^2$; $IM_I = (IM_I^1 - IM_I^1(BC^1)) \cup (IM_I^2 - IM_I^2(BC^2))$; $OM_I = (OM_I^1 - OM_I^1(BC^1)) \cup (OM_I^2 - OM_I^2(BC^2))$; $A = A^1 \cup A^2$; $IP_I = \{ip \mid \forall i \in IM_I, \exists ip \in IP, imessage(i) = ip\}$; $OP_I = \{op \mid \forall o \in OM_I, \exists op \in OP, omessage(o) = op\}$; $P_I = \{p \mid \forall i \in IM_I, \forall o \in OM_I, \exists p \in P, iport(i) = p \vee oport(o) = p\}$; and CCS($BC$) $=$ (CCS(BC^1)| CCS(BC^2))$[f]$, where $f : IM_I^1 \to \mathcal{L}, IM_I^2 \to \mathcal{L}, OM_I^1 \to \mathcal{L}, OM_I^2 \to \mathcal{L}$. We denote the set of input messages $IM_I^1(BC^1) = \{i \in IM_I^1 \mid \exists o \in OM_I^2, f(i) = f(o)\}$ and $IM_I^2(BC^2) = \{i \in IM_I^2 \mid \exists o \in OM_I^1, f(i) = f(o)\}$.

We denote the set of output messages $OM_I^1(BC^1) = \{o \in OM_I^1 \mid \exists i \in IM_I^2, f(i) = f(o)\}$ and $OM_I^2(BC^2) = \{o \in OM_I^2 \mid \exists i \in IM_I^1, f(i) = f(o)\}$. $IM_I^1(BC^1)$ is the set of input messages, received by design component 1, which are sent out by design component 2 from the set of output messages $OM_I^2(BC^2)$. Therefore, the sets of messages $IM_I^1(BC^1)$ and $OM_I^2(BC^2)$ become internal messages that are not visible outside the integration of design component 1 and design component 2. They will not be included in the interface of the integration. Similarly, the sets of messages $IM_I^2(BC^2)$ and $OM_I^1(BC^1)$ will not be included in the interface of the integration either.

We denote this process of behavioral integration of BC^1 and BC^2 by CCS(BC^1, BC^2). Their corresponding implementation of XL processes are BC_1^{XL} and BC_2^{XL}, respectively. Therefore, the integration of BC_1^{XL} and BC_2^{XL} is defined by $BC_1^{XL} \mid BC_2^{XL}$.

A.3. Implementation of sequence properties

The following is the implementation of the two sequence properties described in Section 5.2.3:

```
% Content::show will be eventually invoked
% when Reference::GoTo is invoked.
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).

% Context::show will be eventually invoked
% when Reference::GoTo is invoked.
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)
```

Copyright © 2005 John Wiley & Sons, Ltd.

```
  \/ [-] formula2(ConcreteContext, Show).

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

A.4. Summary of notation and symbols

For the following definitions assume that P and Q are predicates, S, T, and U are sets, X and Y are arbitrary types, and R and Z are relations.

Logic

$P \wedge Q$ conjunction
$P \vee Q$ disjunction
$P \implies Q$ implication
$P \equiv Q$ equivalence
$\exists$ exists
$\forall$ forall

Set theory

$\{x1, x2, \ldots, xn\}$ set enumeration
$\in$ membership
$\notin$ non-membership
$\subset$ subset
Φ empty set
$\{x : S \mid P\}$ the elements of S that satisfy P
$S \cap T$ the intersection of S and T
$S \cup T$ the union of S and T

Prolog

Constant symbols: strings starting with a lowercase alphabetic character or a numeric character.
Variable symbols: strings starting with an uppercase alphabetic character.
Function symbols: strings starting with a lowercase alphabetic character.
Terms: a constant and a variable is a term. A function $f(t_1, \ldots, t_n)$ is a term if $t_1, \ldots, t_n$ are terms.
Atoms: $p(t_1, \ldots, t_n)$ is an atom where p is a predicate symbol and $t_1, \ldots, t_n$ are terms.
Clauses: $p : -q_1, \ldots, q_n$ is a clause where $p, q_1, \ldots, q_n$ are atoms.

CCS

. sequential composition
$+$ non-deterministic choice

Copyright © 2005 John Wiley & Sons, Ltd.

| parallel composition
[ ] relabeling

μ-calculus

μ least-fixed point
ν greatest-fixed point

REFERENCES

1. Nierstrasz O, Dami L. Component-oriented software technology. *Object-Oriented Software Composition*, Nierstrasz O, Tsichritzis D (eds.). Prentice-Hall: Engelwood Cliffs, NJ, 1995; 3–28.
2. Keller RK, Schauer R. Design components: Towards software composition at the design level. *Proceedings of the 20th International Conference on Software Engineering*, 1998; 302–311.
3. Garlan D, Monroe RT, Wile D. Acme: Architectural description of component-based systems. *Foundations of Component-Based Systems*, Leavens G, Sitaraman M (eds.). Cambridge University Press: Cambridge, 2000; 47–67.
4. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading, MA, 1995.
5. Alencar P, Cowan D, Dong J, Lucena C. A pattern-based approach to structural design composition. *Proceedings of the IEEE 23rd Annual International Computer Software and Applications Conference (COMPSAC)*, Phoenix AZ, October 1999; 160–165.
6. Woolf B. Framework development using patterns. *Implementing Application Frameworks*, Fayad M, Schmidt D, Johnson R (eds.). Wiley: New York, 1999; 621–628.
7. Dong J, Alencar P, Cowan D. A behavioral analysis and verification approach to pattern-based design composition. *International Journal of Software and Systems Modeling* 2004; **3**(4):262–272.
8. Clarke E, Grumberg O, Peled DA. *Model Checking*. MIT Press: Cambridge, MA, 1999.
9. Milner R. Communication and concurrency. *International Series in Computer Science*. Prentice-Hall: Englewood Cliffs, NJ, 1989.
10. Ramakrishna YS, Ramakrishnan CR, Ramakrishnan IV, Smolka SA, Swift T, Warren DS. Efficient model checking using tabled resolution. *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, Haifa Israel, July 1997 (*Lecture Notes in Computer Science*, vol. 1243). Springer: Berlin, 1997; 143–154.
11. Kozen D. Results on the Propositional μ-calculus. *Theoretical Computer Science* 1983; **27**:333–354.
12. Stirling C. *An Introduction to Modal and Temporal Logics for CCS* (*Lecture Notes in Computer Science*, vol. 491). Springer: Berlin, 1991; 1–20.
13. Pemmasani G, Guo H-F, Dong Y, Ramakrishnan CR, Ramakrishnan IV. Online justification for tabled logic programs. *Proceedings of the International Symposium on Functional and Logic Programming*, April 2004.
14. Lloyd JW. *Foundations of Logic Programming*. Springer: New York, 1984.
15. Dong J. Design component contracts: model and analysis of pattern-based composition. *PhD Thesis*, Computer Science Department, University of Waterloo, June 2002.
16. Bieber M, Vitali F. Toward support for hypermedia on the World Wide Web. *IEEE Computer* 1997; **30**(1):62–70.
17. Isakowitz T, Stohr EA, Balasubramanian P. RMM: A methodology for structured hypermedia design. *Communications of the ACM* 1995; **38**(8):34–44.
18. Garzotto F, Schwabe D, Paolini P. HDM—a model based approach to hypermedia application design. *ACM Transactions on Information Systems* 1993; **11**(1):1–26.
19. Schwabe D, Rossi G. The object-oriented hypermedia design model. *Communications of the ACM* 1995; **38**(8):45–46.
20. Lange D. An object-oriented design method for hypermedia information systems. *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, 1994.
21. Rossi G, Schwabe D, Garrido A. Design reuse in hypermedia applications development. *Proceedings of the ACM International Conference on Hypertext*, 1997; 57–66.
22. Germán DM, Cowan DD. Towards a unified catalog of hypermedia design patterns. *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, January 2000.
23. Discenza A. Design patterns for WWW museum hypermedia. *Technical Report 99.4*, Politecnico di Milano, 1999.
24. NGA. The National Gallery of Art. http://www.nga.gov/help/help.htm [2002].

Copyright © 2005 John Wiley & Sons, Ltd.

25. Germán DM. Hadez: A framework for the specification and verification of hypermedia applications. *PhD Thesis*, Computer Science Department, University of Waterloo, 2000.
26. Dong J. Adding pattern related information in structural and behavioral diagrams. *International Journal of Information and Software Technology (IST)* 2004; **46**(5):293–300.
27. Dong J, Alencar P, Cowan D. A behavioral analysis approach to pattern-based composition. *Proceedings of the 7th International Conference on Object-Oriented Information Systems (OOIS)*. Springer: Berlin, 2001; 540–549.
28. Lamport L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering* 1977; **3**(2):125–143.
29. Bradfield JC, Stirling C. Modal logic and μ-calculi: An introduction. *Handbook of Process Algebra*, Ponse A, Bergstra JA, Smolka SA (eds.). Elsevier: Amsterdam, 2001; 293–330.
30. Bradfield JC. Fixpoint alternation: Arithmetic transition systems and the binary tree. *Theoretical Informatics and Applications* 1999; **33**(4/5):341–356.
31. Clarke EM, Emerson EA, Sistla AP. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* 1986; **8**(2):244–263.
32. Emerson EA. Automated temporal reasoning about reactive systems. *Logics for Concurrency: Structure versus Automata* (*Lecture Notes in Computer Science*, vol. 1043), Moller F, Birtwistle G (eds.). Springer: Berlin, 1996; 41–101.
33. Emerson EA, Lei CL. Efficient model checking in fragments of the mu-calculus. *IEEE Symposium on Logic in Computer Science (LICS)*, Cambridge, MA, June 1986.
34. Wang W, Hidvégi Zoltán, Bailey AD, Whinston AB. E-process design and assurance using model checking. *IEEE Computer* 2000; **33**(10):48–53.
35. Meyer B. Applying ‘design by contract’. *IEEE Computer* 1992; (October):40–51.
36. Helm R, Holland IM, Gangopadhyay D. Contracts: Specifying behavioral compositions in object-oriented systems. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 1999; 169–180.
37. Beugnard A, Jézéquel J-M, Plouzeau N, Watkins D. Making components contract aware. *IEEE Computer* 1999; **32**(7):38–45.
38. Borgida A, Devanbu P. Adding more ‘dl’ to idl: Towards more knowledgeable component inter-operability. *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999; 378–387.
39. Alencar P, Cowan D, Lucena C. A formal approach to architectural design patterns. *Proceedings of the 3rd International Symposium of Formal Methods Europe (FME)*, 1996; 576–594.
40. Mikkonen T. Formalizing design pattern. *Proceedings of the 20th International Conference on Software Engineering*, 1998; 115–124.
41. Garlan D, Khersonsky S. Model checking implicit-invocation systems. *Proceedings of the 10th International Workshop on Software Specification and Design (IWSSD)*, November 2000; 23–30.
42. Basu S, Smolka SA, Ward OR. Model checking the Java meta-locking algorithm. *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, April 2000; 342–350.
43. Corbett JC, Dwyer MB, Hatcliff J, Laubach S, Pasareanu C, Robby, Zheng H. Bandera: Extracting finite-state models from Java source code. *Proceedings of the 22nd International Conference on Software Engineering*, 2000.
44. McMillan KL. Symbolic model checking: An approach to the state explosion problem. *PhD Thesis*, Carnegie Mellon University, CMU-CS-92-131, 1992.
45. Holzmann GJ. The model checker SPIN. *IEEE Transactions on Software Engineering* 1997; **23**(5):279–295.
46. Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language User Guide*. Addison-Wesley: Reading, MA, 1999.
47. Warmer JB, Kleppe AG. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley: Reading, MA, 1998.
48. Kim S-K, Carrington D. Formalizing the UML class diagram using Object-Z. *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML)* (*Lecture Notes in Computer Science*, vol. 1723). Springer: Berlin, 1999; 83–98.
49. Abdurazik A, Offutt J. Using UML collaboration diagrams for static checking and test generation. *Proceedings of the 3rd International Conference on the Unified Modeling Language (UML)* (*Lecture Notes in Computer Science*, vol. 1939). Springer: Berlin, 2000; 383–395.

Copyright © 2005 John Wiley & Sons, Ltd.