

A Formal Framework for Design Component Contracts

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

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

Abstract – *Building large software systems out of existing software components can save time and cost. These software components range from architectural and design components to binary components in different phases of software development. Component technologies lead to increasing productivity and flexibility. 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. In this paper, we concentrate on the pattern-based design components and their compositions, which are captured as contracts through a composition theory. More specifically, design component contract is defined based on process calculus and logic programming; the properties that related to the design component contract are captured, and tools are used to automatically verify these properties. This research will enhance the capabilities of formal system modeling and analysis by providing a rigorous basis for high-assurance integration of diverse pattern-based components.*

Keywords: Component-Based Design, Process Calculus, Contract, Design Pattern, Model Checking

1 Introduction

Component-based approaches have been proposed to create and deploy software systems assembled from components [22, 16, 27, 11]. These approaches focus on building software systems by assembling prefabricated, configurable, and independently evolving building blocks. Use of previously developed components in building software systems is an appealing idea because of the apparent reduction in cost and effort. Use of components should also lead to faster time-to-market for complex software applications. Further, since these components have probably been tested in use and may have even been formally validated their combination should produce a more robust software system. Therefore, component-based software development is a promising solution to some of the problems that designers, developers and integrators face when building their systems. However, unanticipated interactions among the components are often the cause of failures. Many experiments [9, 15] support this view and indicate that deep knowledge about the domain and

about the software architecture and design is a critical factor in the construction and integration of such applications.

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 pattern-based approach is that it may greatly simplify the construction of software systems, reuse experience and reduce cost. Design components [16], such as design patterns [8], have been proposed to reify good design practice from conceptual design building blocks into a tangible and composable form. Design components focus on component-based problem solving instead of component-based implementation. There has been substantial interest in discovering and documenting such reusable design experience in different domains as, for example, in hypertext design [24]. However, there has been little work on reasoning about the interactions among the design patterns when they have been composed.

A design component is often characterized by some properties relating to its structure and behavior. Nevertheless, because of the undesired interactions with other components, some properties about this component may not hold after composition. Ensuring such properties still hold after the composition can increase our confidence in the correctness and reliability of the integration. Analyses based on formal, rather than informal, techniques make our reasoning precise; moreover, they are amenable to mechanical aids such as syntax and semantics checkers. The analysis of composition properties should be performed at the design level, because errors in the design of these systems are difficult and expensive to find and correct if propagated to the implementation phase [25].

Design analysis can assist in discovering bugs in a design early in the development phase and reduce the cost of finding and correcting them downstream [29]. In this paper, we introduce a rigorous modeling and analysis approach to software design composition based on formal specification and automated verification techniques. The approach involves the modeling of design components and their composition, and a framework in which design compositions can be analyzed. We characterize the structural and behavioral aspects,

and a specific form of evolution of these components. 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 trade-offs of design decisions and choose appropriately.

The rest of this paper is organized as follows. We provide an overview of the approach in the next section. In Section 3, we define the design component contract and concentrate on the behavioral aspect. In Section 4, we apply our approach to the analysis of hypermedia web-based applications. In the last two sections, we discuss the related work and conclude this paper.

2 Overview of the Approach

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

A design component contract includes structural contract and behavioral contract. Structural contract is modeled in predicate logic, whereas the behavioral contract is modeled in Calculus of Communicating Systems (CCS) [20]. Both contracts include three kinds of operations: instantiation, integration and evolution. Since each contract defines the generic information about a design component, the instantiation operation can be used to apply a generic contract in a particular application. The integration operation formally defines how to compose two or more contract to form a new contract. As each design component is not fixed, it often evolves in some restricted ways. The evolution operation formally defines these kinds of changes. The abstract specifications of the formal models of design components are described, in details, in Section 3.

Four different kinds of properties can be analyzed: structural, behavioral, hybrid and evolutionary properties. The structural properties describe the relations of the constructs of each design component, such as connectivity of classes by inheritance or association relations in object-oriented systems. The structural properties contain atom, consistency, integrity and link properties. The behavioral properties are constraints such as safety, liveness, event ordering, and action sequence of each design component. The behavioral properties consist of global, consistency, concurrency, se-

quence and occurrence properties. A combination of structural and behavioral properties is a hybrid property. The properties related to system evolution are evolutionary properties, which describe the possible structure changes to adapt new requirements of each design component.

The structural contract is implemented in Prolog. The behavioral contract is implemented in XL that is the model specification language of XMC [23]. Similarly, the structural and evolutionary properties are implemented in Prolog. The behavioral properties are implemented in μ -calculus [17, 26]. The hybrid properties are implemented in Prolog and μ -calculus. XSB Prolog and XMC model checker are used as verification tools to check different properties against their corresponding models. The analysis results show either property verified or counter examples.

In the next sections, we will only show some details about the formal model of design component, an implementation of the contract model in Prolog and XMC, some selected properties, and a short description of a case study with some analysis results. For interested readers, we refer to [5] for more details.

3 Design Component Contracts

A design component contract is a formal model of 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 calculus, respectively. This meta-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 design component is modeled based on logic program. A structural contract is defined by constant symbols, variable symbols, and predicate symbols. A meta-model of structural contract is instantiated, evolved, and integrated with other contracts in a particular application.

Behavioral Contract: the behavioral aspect of design component is modeled based on process calculus. 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: instantiation, integration, evolution. As each contract is a meta-model of a design component, the application of the component requires instantiating these generic definitions with application dependent information. The composition of 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. Each component is modeled with changes in mind in that it describes expected evolution path in terms of addition or removal of certain parts of the component.

Property: each design component possesses certain properties that characterize it from others. 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 design component contract as clauses in a normal logic program [18] and the behavioral aspect of design component contract by a process calculus, called Calculus of Communicating Systems (CCS) [20]. Meanwhile, we express structural properties by logic rules and behavioral properties by μ -calculus [17]. This allows us to formally check properties about the compositions of design components. In the remainder of this section, we formally define the semantics of the behavioral aspects of a design component contract and the integration operation¹.

3.1 Behavioral Contracts

The behavioral contract describes the dynamic information, such as the collaboration among the objects participating in the component and the creation of new objects. The behavioral contract is modeled by the collaborations of societies of objects that play different roles and work together to carry out some behavior that is bigger than the sum of the elements. The behavioral contract is essential because the structural contract only captures the static information, but patterns are also characterized by the interactions among the objects and operations.

We have chosen process calculus for defining a formal semantic model of behavioral contracts due to their powerful model of behavior and concurrency. Process calculus allows for hierarchical description of processes and they are amendable for verification, analysis and compositional reasoning. The particular process calculus we have chosen for modeling the semantics of behavioral contracts is CCS. The reasons for choosing it include that it is a powerful modeling language and it is well supported with a model checking tool.

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

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

where terms generated by P are also called processes; the prefixing $a.P$ is sequential composition and the action a range over a nonempty set of actions including a distinguished action τ for unobservable activities; the summation

¹We omit the definitions of the structural contract, the instantiation and evolution operations, the modeling rules, and the definitions of structural, behavioral and evolutionary properties. We refer to [5] for details.

$P + P$ is non-deterministic choice and $\sum_{i \in I} P_i$ is the summation over index set I ; $P|P$ is parallel composition and $\prod_{i \in I} P_i$ is the parallel composition over index set I ; $P[f]$ is relabelling where f are relabeling functions preserving observability (i.e., $f^{-1}(\tau) = \{\tau\}$); 0 is nil process that cannot execute any action. We shall often omit the dot of the action prefix, and drop the trailing 0 's from expressions, therefore for example, $a.0 + b.c.0$ is shorthanded to $a + bc$. We refer to [20] for further details of CCS.

Definition 3.1 (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. The behavior of each object $o \in O$ (O is the set of objects in the system) is modeled as a process $p \in P$ in CCS, where $P : O \leftrightarrow P$. We denote the process for object $o \in O$ by $P(o)$.*
- *IP is a finite set of input ports attached to a process. A process can input messages from its input ports, where $iport : IP \rightarrow P$. We denote the set of input ports for the process $p \in P$ by $IP(p) = \{i \in IP \mid iport(i) = p\}$.*
- *OP is a finite set of output ports attached to a process. A process can output messages from its output ports, where $oport : OP \rightarrow P$. We denote the set of output ports for the process $p \in P$ by $OP(p) = \{o \in OP \mid oport(o) = p\}$.*
- *IM is a finite set of input messages sent to a process, where $imessage : IM \rightarrow IP$. We denote the set of input message sent to the process $p \in P$ from other processes by $IM(p)$. These input messages are received from one or more input ports of the process p . We denote the set of messages received via input port $ip \in IP$ of process $p \in P$ by $IM(ip, p) = \{i \in IM \mid ip \in IP(p), imessage(i) = ip\}$.*
- *OM is a finite set of output messages sent from a process, where $omessage : OM \rightarrow OP$. We denote the set of output message sent from the process $p \in P$ to other processes by $OM(p)$. These output messages are sent out from one or more output ports of the process p . We denote the set of messages sent via output port $op \in OP$ of process $p \in P$ by $OM(op, p) = \{o \in OM \mid op \in OP(p), omessage(o) = op\}$.*
- *IM_I is the finite set of input messages sent from outside the design component to a process $p \in P$, denoted by $IM_I(p)$ and $IM_I(p) \subset IM(p)$.*
- *OM_I is the finite set of output messages sent outside the design component from a process $p \in P$, denoted by $OM_I(p)$ and $OM_I(p) \subset OM(p)$.*
- *A is a finite set of actions that can be performed by a process. We denote the set of actions of the process $p \in$*

P by $A(p)$. An action is used to model a method of an object.

Definition 3.2 (CCS-Process of Behavioral Contract)

Let $BC = \langle P, IP, OP, IM, OM, IM_I, OM_I, A \rangle$ 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,

$$CCS(BC) = (\prod_{p \in P} BC(p))[f].$$

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

Definition 3.3 (Relabelling) Let \mathcal{L} be a set of labels. The relabelling function f is defined by $f : IM - IM_I \rightarrow \mathcal{L}$ and $f : OM - OM_I \rightarrow \mathcal{L}$. For each $i \in (IM - IM_I)$ and $o \in (OM - OM_I)$, iff $f(i) = f(o)$, i.e. message i and message o are relabeled to the same label, then these messages synchronize their corresponding processes. This relabelling function can be abbreviated by $b_1/a_1, \dots, b_n/a_n$ so that the f renames a_i to b_i ($f(a_i) = b_i, a_i \neq b_i$), and leaves any other action ($a_j = b_j, j \notin [1..n]$) unchanged. Associated with f is the relabelling operator $[f]$.

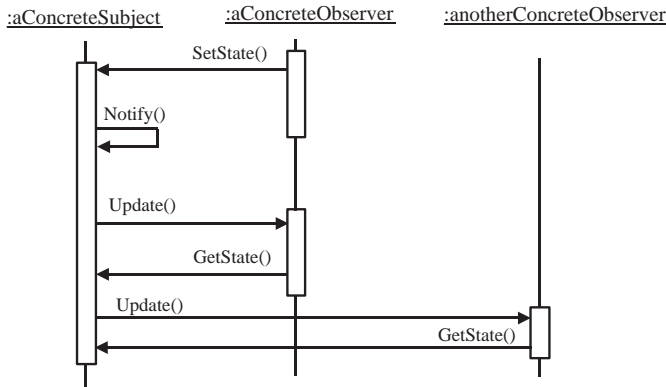


Figure 1: Interaction Diagram for Observer Pattern

Consider the behavior of the Observer pattern [8]. Component behaviors include the interactions, shown in Figure 1, among the constituted objects of the components and the state changes within these objects, shown in Figure 2 and 3. In this example, the inter-object relationships of the Observer pattern component describe two observers attached to one subject. When one observer changes its state, it sends a message (SetState) to let the subject change its state. The subject, then, notify all attached observers to update their

own states by calling back the GetState operation. All participants of this interaction have their own state changes. For example, each observer (see Figure 2) can attach or detach itself from a subject. Whenever its state has changed, it can let the corresponding subject know the change. It also listens to all update requests from the subject and changes its state accordingly. Similarly, a subject (see Figure 3) can attach or detach a number of observers on its states. Whenever it receives a request to change its states, it will change its states and notify all attached observers to update their states.

Example 3.1 Let $BC_{Observer} = \langle P, IP, OP, IM, OM, IM_I, OM_I, A \rangle$ be the behavioral contract of observer pattern component and its CCS processes be $CCS(BC_{Observer})$, where

- The set of processes in the design component is $P = \{aConcreteSubject, aConcreteObserver, anotherConcreteObserver\}$.
- The set of input ports is $IP = \{O2S, S2O, Ntfy, Input\}$, where $O2S$ and $S2O$ stand for input ports from the Observer process and the Subject process, respectively, and $Ntfy$ is an input port for message Notify. All messages from outside of the process are received in the input port $Input$.
- The set of output ports is $OP = \{O2S, S2O, Ntfy\}$, where $O2S$ and $S2O$ stand for output ports to the Subject process and the Observer process, respectively, and $Ntfy$ is an output port for message Notify.
- The set of input messages is $IM = \{Attach, Detach, SetState, GetState, Update, Notify, Change\}$.
- The set of output messages is $OM = \{Attach, Detach, SetState, GetState, Update, Notify\}$.
- The set of input messages in the interface is $IM_I = \{Change\}$.
- The set of output messages in the interface is $OM_I = \{\}$.
- The set of actions is $A = \{Attach, Detach, SetState, GetState, Update, Notify, Change\}$
- The CCS processes are defined by $CCS(BC_{Observer})$, which is implemented in XL as follows²:

```

ObserverBehavior ::= Subject(aConcreteObserver)
  | Observer(aConcreteObserver)
  | Observer(anotherConcreteObserver)
Observer(Name) ::= out(Attach) o Observer
  # out(Detach) o Observer
  # in(Change) o action(Change) o out(SetState) o Observer
  # in(Update) o action(Update) o out(GetState) o Observer
Subject(Name) ::= in(Attach) o action(Attach) o Subject
  # in(Detach) o action(Detach) o Subject
  # in(SetState) o action(SetState)
  o out(Notify) o Notifying
Notifying ::= in(Notify) o action(Notify)
  o out(Update) o Updating
Updating ::= in(GetState) o action(GetState) o Subject

```

²Note that, in XL, the sequential composition is represented by “o” instead of “;”, the non-deterministic choice is represented by “#” instead of “+”. Other operators are the same as those in CCS.

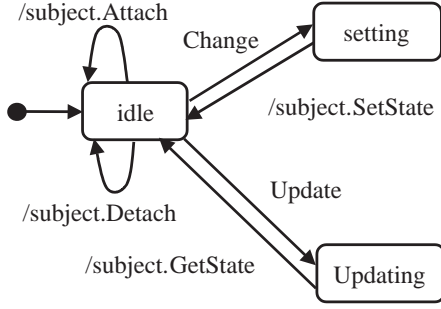


Figure 2: State Diagram for Observer

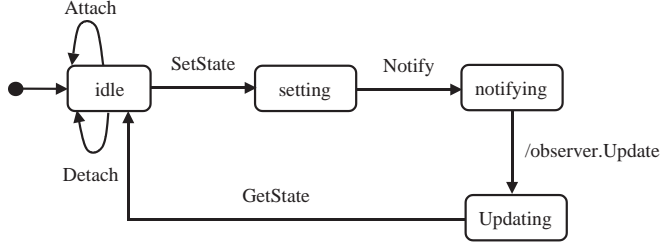


Figure 3: State Diagram for Subject

3.2 Behavioral Integration

Definition 3.4 (Behavioral Interfaces) Let $BC = \langle P, IP, OP, IM, OM, IM_I, OM_I, A \rangle$ be the behavioral contract of a design component. The interface of behavioral contract is a tuple $BI = \langle IM_I, OM_I, IP_I, OP_I, P_I \rangle$, $BI \subset BC$, where

- IM_I is the set of input messages sent from outside the design component. It is defined in BC .
- OM_I is the set of output messages sent outside the design component. It is defined in BC .
- IP_I is the set of input ports that are used to receive input messages in the set IM_I . Thus, $IP_I = \{ip \mid \forall i \in IM_I, \exists ip \in IP, imessage(i) = ip\}$.
- OP_I is the set of output ports that are used to send output messages in the set OM_I . Thus, $OP_I = \{op \mid \forall o \in OM_I, \exists op \in OP, omessage(o) = op\}$.
- P_I is the set of processes that receive (send) input (output) messages in the set IM_I (OM_I). Thus, $P_I = \{p \mid \forall i, o, i \in IM_I, o \in OM_I, \exists p \in P, iport(i) = p \vee oport(o) = p\}$.

Definition 3.5 (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

- $CCS(BC) = (CCS(BC^1) \mid CCS(BC^2)) [f]$, where $f : IM_I^1 \rightarrow \mathcal{L}, IM_I^2 \rightarrow \mathcal{L}, OM_I^1 \rightarrow \mathcal{L}, OM_I^2 \rightarrow \mathcal{L}$. 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 $BC_1^{XL} \mid BC_2^{XL}$.

Informally, the integration of behavioral contracts is defined based on the integration of these contracts and the corresponding CCS processes. More specifically, the process names (P), input/output ports (IP/OP), input/output messages (IM/OM) and actions (A) are combined through set union. The input messages (IM_I) that are in the interface of the integration are defined as a union of the input messages that are in the interface of each individual contract except those input messages that are sent out from the components within the integration. The output messages (OM_I) that are in the interface of the integration are defined similarly. In other words, part of the interface of each component may become internal to the integration. The integration of the CCS-processes ($CCS(BC)$) is the parallel composition of the CCS-processes of each component with a new relabelling function.

4 Analyzing Hypermedia Web-Based Applications

Large hypermedia applications often contain many design patterns working in concert to solve complex problems.

Many problems are ubiquitous and designers are likely to face them eventually. A catalog of fifty one hypermedia design patterns have been documented in [13]. Discenza [4] has conducted a study of hypermedia design patterns based on thirty four 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 analyzed the design of a virtual art museum (National Gallery of Art, <http://www.nga.gov/>), which is a complex application containing more than 100,000 objects in the Gallery's collection database [21]. A detailed model and description of this application can be found in [12, 4]. 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 particular, we focused the analysis on the design of a virtual art museum that has many collections of paintings. The design includes the applications of several hypermedia design patterns, such as the Navigational Contexts pattern, the Active Reference pattern, the Navigational Observer pattern, and the Information on Demand pattern [24]. In this case study, we first built formal models of the structural and behavioral aspects of these patterns into structural and behavioral contracts, respectively. We, then, described the instantiation, integration and evolution operations of these contracts. We, therefore, provided an implementation of the structural and behavioral contracts in XSB Prolog and XMC, respectively. We have also characterized different kinds of properties that can be analyzed within our framework. We give here some examples of properties that can be checked in our framework. For example, a deadlock property, which describes a deadlock system state is reachable, is defined as follows:

Example 4.1 (Deadlock) Consider the behavioral contract $BC_{Combine}$ of the integration of all behavioral contracts of this design. The absence of deadlock can be denoted by $\nu X.[-]X \wedge \langle - \rangle tt$ in μ -calculus. The implementation of these properties in XMC is shown as following:

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

A test Prolog program is written as following:

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

test :-
    write('Freedom of deadlock'),
    checkit(mck(combination,deadlock)).
```

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

```
| ?- test.
Freedom of deadlock    mck(combination,deadlock) is true.

yes
```

where `mck` checks the properties `deadlock` against the model `combination`. The results show that the integration of the design components is freedom of deadlock. The analysis of concurrency properties is useful in many applications. For example, system deadlock may lead to denial of service (DOS) in E-commerce systems [28].

As another example, we demonstrate the verification of an even-length path property which describes that there exists a maximal path of even length of an action.

Example 4.2 (Even length path) Consider the behavioral contract $BC_{Combine}$ of the integration of all behavioral contracts of this design. 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 made twice, not interleaved with another request, in order for the history being reset. The property that there exists an even-length of action `reset` can be denoted by $\mu X.[reset]ff \vee \langle reset \rangle \langle reset \rangle X$ in μ -calculus. The implementation of these properties in XMC is shown as following:

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

A test Prolog program is written as following:

```
:- 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 following:

```
| ?- 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 pairs. The reason for this error is that the original specification of the behavioral contract of the Navigational Observer pattern (one of the patterns applied in this design) does not require double requests for a history reset.

In summary, we have analyzed many different kinds of properties, including structural, behavioral and evolutionary ones [5, 7]. Our experience showed that it is highly beneficial to apply logic programming techniques for modeling and analysis of software design components and their compositions.

5 Related Work

The notion of contracts in software development is attributed to Meyer [19]. Another contribution, the OO-contracts of Helm et al. [14] focus on specifying the behavior and interactions between objects in a system. Helm et al. noticed that the behavior of an object could not be inferred from its interface, leading to design and reuse problems. Contracts formalize the behavioral relationship between objects and define a set of participants and their obligations. In [2], Beugnard et al. propose a four-level contract for components to increase trust. In [3], 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 contract and a rigorous analysis approach to software design composition based on automated verification techniques.

Keller et al. [16] described a methodical approach to design composition which was illustrated as a process within a four-dimensional design space. They characterized a special kind of component, called design component, and discussed a development process to compose these components at the design level and generate source code frames or executable code. Although our approach is also in the area of software composition, it focuses on the formal, declarative, and property-based aspects of design composition.

Model checking implicit-invocation systems has been investigated in [10]. Implicit-invocation systems were modeled by the structure elements including components, event types, shared variables, event bindings, event delivery policies and concurrency models. A run-time state model was also constructed with the mechanisms that handle event announcement, event buffering, and method invocation and the mechanisms that implement event dispatch and event delivery policy. This abstraction and modeling process is highly domain-specific. Model checking Java meta-locking algorithm has also been conducted in [1]. 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.

6 Conclusions

In this paper, we have introduced a formal model of design component based on contract and a rigorous analysis approach to software design composition based on automated verification techniques. Our approach has several advantages. First, it allows us to find errors in the design composition early in the development process and save the costs of having to correct them later. Second, it provides mechanisms to achieve automated verification of the properties of software designs. Third, the generic representations of design components can be stored in a repository and retrieved for instantiation and integration in a specific application. Fourth, as the composition of components can be treated as a component, the design analysis can scale up incrementally to large component-based software systems. Fifth, contracts were

able to capture the complex design component topologies and interactions and could be used to analyze pattern-based designs.

Besides verifying structural, behavioral and evolutionary properties, we are characterizing properties that we can use in our analysis of design compositions. These classes may include properties about (real-)time and access control. Techniques can also be provided to map the XL counter-examples back onto the original structural and behavioral descriptions to assess how much of this feature can be automated and how much effort is required to be able to show which parts of the structural and behavioral descriptions need to be revised under the light of the counter-examples.

The analysis results shown in this paper indicate that it could be highly beneficial to pursue further research on how the structural design changes may affect the behavior of software system composition, and how these changes can affect various properties of the design composition. These analyses will provide guidance on important design decisions, but also be documented to provide information on the rationale behind these decisions [6, 7].

References

- [1] S. Basu, S. A. Smolka, and O. R. Ward. Model Checking the Java Meta-Locking Algorithm. *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, pages 342–350, April 2000.
- [2] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.
- [3] A. Borgida and P. Devanbu. Adding more “dl” to idl: Towards more knowledgeable component interoperability. *Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA*, pages 378–387, May 1999.
- [4] A. Discenza. Design Patterns for WWW Museum Hypermedia. *Technical Report 99.4, Politecnico di Milano*, 1999.
- [5] J. Dong. Design Component Contracts: Model and Analysis of Pattern-Based Composition. *Ph.D. Thesis, Computer Science Department, University of Waterloo*, June 2002.
- [6] J. Dong, P. Alencar, and D. Cowan. Ensuring Structure and Behavior Correctness in Design Composition. *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS), Edinburgh UK*, pages 279–287, April 2000.
- [7] J. Dong, P. Alencar, and D. Cowan. A Behavioral Analysis Approach to Pattern-Based Composition. *Proceedings of the 7th International Confer-*

- ence on *Object-Oriented Information Systems (OOIS)*, Springer-Verlag, Calgary, Canada, August 2001.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.
- [9] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts. *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, April 1995.
- [10] D. Garlan and S. Khersonsky. Model Checking Implicit-Invocation Systems. *Proceedings of the 10th International Workshop on Software Specification and Design (IWSSD)*, pages 23–30, November 2000.
- [11] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems, (Leavens G. and Sitaraman M., eds.) Cambridge University Press*, pages 47–67, 2000.
- [12] D. M. Germán. Hadez: A Framework for the Specification and Verification of Hypermedia Applications. *Ph.D. Thesis, Computer Science Department, University of Waterloo*, 2000.
- [13] D. M. Germán and D. D. Cowan. Towards a Unified Catalog of Hypermedia Design Patterns. *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, January 2000.
- [14] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA), USA*, pages 169–180, October 1990.
- [15] S. A. Hissam. Experience Report: Correcting System Failure in a COTS Information System. *Proceedings of the International Conference on Software Maintenance, Bethesda, USA*, pages 170–176, November 1998.
- [16] R. K. Keller and R. Schauer. Design Components: Towards Software Composition at the Design Level. *Proceedings of the 20th International Conference on Software Engineering*, pages 302–311, 1998.
- [17] D. Kozen. Results on the Propositional μ -calculus. *Theoretical Computer Science*, 27:333–354, 1983.
- [18] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, New York, NY, 1984.
- [19] B. Meyer. Applying “design by contract”. *IEEE Computer*, pages 40–51, October 1992.
- [20] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
- [21] NGA. The National Gallery of Art. Available from <http://www.nga.gov/help/help.htm>, 2002.
- [22] O. Nierstrasz and L. Dami. Component-Oriented Software Technology. *Object-Oriented Software Composition, ed. O. Nierstrasz and D. Tschritzis, Prentice Hall*, pages 3–28, 1995.
- [23] Y. Ramakrishna, C. Ramakrishnan, I. Ramakrishnan, S. Smolka, T. Swift, and D. Warren. Efficient Model Checking Using Tabled Resolution. *Proceedings of the 9th International Conference on Computer Aided Verification (CAV), Haifa Israel, LNCS1243, Springer-Verlag*, pages 143–154, July 1997.
- [24] G. Rossi, D. Schwabe, and A. Garrido. Design Reuse in Hypermedia Applications Development. *Proceedings of the ACM International Conference on Hypertext*, pages 57–66, April 1997.
- [25] J. M. Rushby. Mechanizing Formal Methods. *Proceedings of the 9th International Conference of Z Users (ZUM95), Jonathan P. Bowen and Michael G. Hinhey (eds), LNCS967, Springer-Verlag, Limerick, Ireland*, pages 105–113, September 1995.
- [26] C. Stirling. An Introduction to Modal and Temporal Logics for CCS. *Lecture Notes in Computer Science 491, Springer-Verlag*, pages 1–20, 1991.
- [27] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Longman, Reading, Mass., 1998.
- [28] W. Wang, Z. Hidvégi, A. D. Bailey, and A. B. Whinston. E-Process Design and Assurance Using Model Checking. *IEEE Computer*, 33(10):48–53, October 2000.
- [29] H. Yu, X. He, Y. Deng, and L. Mo. Formal Analysis of Real-Time Systems with SAM. *Proceedings of the 4th International Conference on Formal Engineering Methods (ICFEM)*, pages 275–286, Oct. 2002.