

Model Checking the Composition of Hypermedia Design Components

Jing Dong
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
jdong@csg.uwaterloo.ca

Abstract

Component-based software development aims at building software system by assembling software components to reduce cost, risk and time-to-market. However, conflicts among components constitute a crucial barrier to successful software composition. In this paper, we present an approach to analyze the properties of components and their compositions in order to detect and correct composition errors. We also demonstrate how model checking can be used to verify properties about the composition of design components. Furthermore, using a hypermedia case study, we show how to represent, instantiate, and integrate design components, and how to find composition errors by applying model checking techniques.

1 Introduction

Large-scale software applications are not usually built from scratch. Instead, most of these applications are developed by reusing building blocks such as interface widgets, design experience, and general design components. Component-based software development [31] aims at building software systems by assembling software components to reduce cost, risk and time-to-market.

However, complex software applications can not normally be built simply by putting software components together. Many experiments [16, 18] indicate that a deep knowledge about the domain and about the software design is

needed to develop such applications. In addition, many software components must guarantee critical functional, fault-tolerant, real-time and performance properties. Proving that such properties still hold after the composition is carried out can increase our confidence about the integration correctness and reliability. Adopting formal techniques, rather than informal approaches, makes our proof process precise. Moreover, these proofs can be mechanized. We should also notice that errors in software designs are difficult to find and expensive to correct if propagated to the implementation phase. Therefore, designers should be able to formally analyze software compositions at the design level.

Design components [20] have been proposed to reify good design practice, such as design patterns [15], from conceptual design building blocks into a tangible and composable form. Design components focus on component-based problem solving rather than on component-based implementation. There has been a substantial amount of work in the area of discovering and documenting reusable design patterns in various domains. Specifically, many patterns have been documented in the area of hypertext design [28]. However, less work has been done on verifying the properties of design components and their compositions in order to detect and correct composition errors.

Model checking has been successfully used by some researchers in the hardware community to verify safety and liveness properties [7, 4, 12]. It has also been used in the software community, e.g., in requirement analysis

[3], in distributed cache coherence analysis [34], and in Java meta-locking algorithm analysis [5]. This paper demonstrates the feasibility of using model checking to analyze the properties of software composition at the design level.

The remainder of this paper is structured as follows. In Section 2, we describe the gist of our approach to analyze design composition properties; the approach has an underlying analysis process based on model checking techniques. In Section 3, we present a hypermedia case study to illustrate how composition properties can be verified using a model checker. In Section 4, we discuss about general applications of formal methods in software engineering. The last two sections focus on related work and conclusions.

2 Analysis Technique

Model checking is a method of verifying algorithmically a formula against a logic model [7]. This verification technique can be automated for some temporal logics. In our case, we assume a logic model Σ representing the software design components and their compositions and a logic formula ϕ representing a property of these components. In order to establish whether the component-based system satisfies the properties, it is checked whether the formula ϕ holds in the model Σ .

2.1 A Model Checker

XMC [26] is a model checker for verifying temporal properties of a system. It is written in the XSB tabled Prolog programming system [35]. Temporal properties are expressed in the alternation-free fragment of the μ -calculus [21, 29]; the system to be verified is described in the specification language for XMC (called XL) which is a highly expressive extension of value-passing CCS [24]. Prolog terms and predicates are used to represent values and computations, respectively. In addition, specifications can make use of recursive data structures and computations. The syntax of XL specification follows:

```
Pdef --> ( Pname ::= Pexp . ) *
Pname --> Prolog Term
```

```
Pexp --> Pexp o Pexp Prefix
| Pexp # Pexp Choice
| Pexp '|' Pexp Parallel Composition
| Pexp @ PortMap Relabelling
| Pexp \ PortList Restriction
| Pname Recursion
| in(Port,Term) Communication (input)
| out(Port,Term) Communication (output)
| Action Communication (non-sync)
| Comp Computation (Prolog expression)
| if(Comp,Pexp,Pexp) Conditional Expression
| zero Empty process (0 in CCS)
| nil Empty computation

PortMap --> [PortTerm/PortTerm (, PortTerm/PortTerm)*]

PortList --> { PortTerm (, PortTerm)* }

Port(Term) --> Prolog Term

Action --> Prolog Atom

Comp --> Prolog Predicate
```

Pname is a parameterized process name, represented as a Prolog term; **Comp** is a computation, e.g., X is $Y+1$; the process $\text{in}(\text{Port}, \text{Term})$ inputs a value over port **Port** and unifies it with term **Term**; $\text{out}(\text{Port}, \text{Term})$ outputs term **Term** over port **Port**; the process $\text{if}(\text{Comp}, \text{Pexp}, \text{Pexp})$ behaves like the first **Pexp** if computation **Comp** succeeds and otherwise like the second **Pexp**. The operation ‘o’ denotes sequential composition; ‘|’ is parallel composition; ‘#’ is nondeterministic choice; ‘@’ is relabeling where **PortMap** is a list of substitutions; and ‘\’ is restriction where **PortList** is a list of port names. Recursion is provided by a set of process definitions, **Pdef**, of the form **Pname ::= Pexp**.

Temporal properties are expressed in the modal μ -calculus whose semantics is usually described over sets of states of labeled transition systems. The μ -calculus is encoded in XMC in an equation form as follows:

```
D --> Z += F (least fixed point)
| Z -= F (greatest fixed point)
F --> Z | tt | ff | F \vee F | F \wedge F | <A> F | [A] F
```

Z is a set of formula variables encoded as Prolog atoms and **A** is a set of actions; **tt** and **ff** are propositional constants; \wedge and \vee are standard logical connectives; $\langle A \rangle F$ denotes that possibly after the action of **A** the formula **F** holds; $[A] F$ denotes that necessarily after the action **A** the formula **F** holds.

Consider, for example, the specification of the Alternating Bit Protocol [32] in XL. We assume that any text after the % character is a comment.

```

medium(Get, Put) ::=
  in(Get, Data);
  { out(Put, Data)
  # action(drop)
  };
medium(Get, Put).

sender(AckIn, DataOut, Seq) ::=
  %% Seq is the sequence number of
  %% the next frame to be sent
  out(DataOut, Seq);
  {
    in(AckIn, AckSeq);
    if AckSeq == Seq
      %% successful ack, next message
      then {
        #Seq is 1-Seq;
        sendnew(AckIn, DataOut, #Seq)
      }
      %% unexpected ack, resend message
      else sender(AckIn, DataOut, Seq)
  }
  #
  %% upon timeout, resend message
  sender(AckIn, DataOut, Seq)
}.

sendnew(AckIn, DataOut, Seq) ::=
  action(sendnew);
  sender(AckIn, DataOut, Seq).

receiver(DataIn, AckOut, Seq) ::=
  %% Seq is the expected next sequence number
  in(DataIn, RecSeq);
  if RecSeq == Seq
    then {
      #Seq is 1-Seq;
      action(rcv);
      out(AckOut, RecSeq);
      receiver(DataIn, AckOut, #Seq)
    }
    else {
      %% unexpected seq, resend ack
      out(AckOut, RecSeq);
      receiver(DataIn, AckOut, Seq)
    }
  }.

abp ::=
  sendnew(R2S_out, S2R_in, 0)
  | medium(S2R_in, S2R_out) % sender -> receiver
  | medium(R2S_in, R2S_out) % receiver -> sender
  | receiver(S2R_out, R2S_in, 0).

%% A packet can be lost without being received
drop_packet += <sendnew>lost \ / <->drop_packet.
lost += <sendnew>tt \ / <-rcv>lost.

%% The system can deadlock.
deadlock += [-] ff \ / <-> deadlock.

```

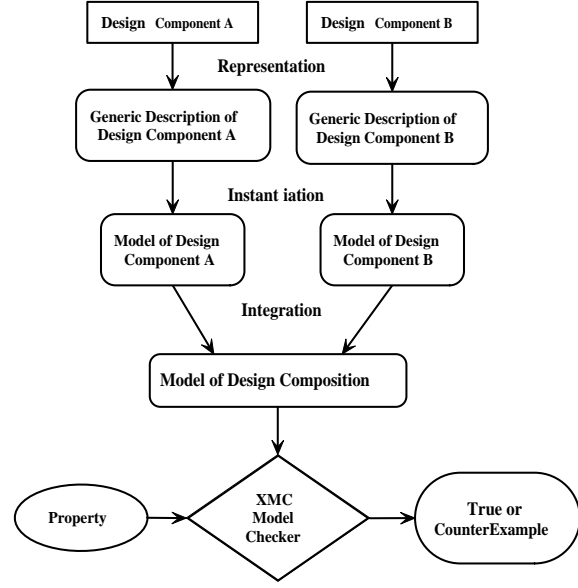


Figure 1: The Design Analysis Process

2.2 A Design Analysis Process

Figure 1 illustrates the main characteristics of the design analysis process underlying our approach. The design components are represented in a declarative way using XL. The component representations are generic in the sense that they capture good design practice in a domain-independent way. These declarative representations, which constitute models of the design components, are instantiated into concrete domain-specific representations and, in this way, design practice can be reused. The concrete design components are integrated to form a model of the design composition, Σ , which is then submitted to a model checker. We use the model checker to check Σ against the property specification ϕ . The model checker outputs either true, if Σ satisfies ϕ , or a counterexample, if it does not.

Initially, design components, such as design patterns, are represented in XL and stored in a XSB Prolog database. There are several advantages of using XSB Prolog as a repository of design knowledge. First, the representation of a component can be reused by instantiating its corresponding generic XL description to produce a concrete domain-specific component representation. Second, the properties and con-

straints of each design component can be described in μ -calculus and thus be checked in XMC. Third, the addition and removal of structural facts about design components can be accomplished by using the Prolog *assert* and *retract* clauses. Fourth, the design component representations can be recovered through the Prolog deductive facilities [22].

Design components are represented in terms of object-oriented design primitives in a predicate-like format. Each design primitive consists of two parts: *name* and *argument*. The *name* part contains the name of an entity or a relation in object-oriented design, such as class, inheritance, etc. The *argument* part contains general information about an entity or a relation such as the information about the participants of an inheritance relation. In the following, we present the syntax and the meaning of the design primitives used in this paper¹:

- `class(C)`: *C* is a class.
- `abstractclass(C)`: *C* is an abstract class.
- `inherit(A, B)`: *B* is a subclass of *A*.
- `variable(C, A, V, T)`: *V* is the name of an attribute in class *C* with type *T*. *T* is optional. *A* describes the access right of this attribute, e.g., public, private, or protected.
- `method(C, A, F, R, P1, T1, P2, T2, ...)`: *F* is a method of a class *C*. *A* describes the access right of this method, e.g., public, private, or protected. *R* describes the return type. If no return value is required *R* can be the value “void”. The method’s parameters and their types are *P*₁, *T*₁, *P*₂, *T*₂, ..., respectively, and are optional. The return type *R* is also optional if the method has no parameters.
- `invoke(C, Cf, O, Of, P)`: A method *O*_{*f*}, which belongs to the object *O*, is invoked in the method *C*_{*f*} of the class *C*, where *P* is the parameter of the method *O*_{*f*}. *P* can contain zero or more parameters depending on the number of parameters the method *O*_{*f*} has.

¹A more comprehensive set of design primitives is described in [10].

- `element(E1, S1, E2, S2, ...)`: *E*₁ is an element of set *S*₁. *E*₂ is an element of set *S*₂, and so on. When universal quantification *forall* and *element* are used together, the quantification is carried out over all the elements of the sets *S*₁, *S*₂, ..., and *S*_{*n*}.

As an example, the XSB Prolog representation of the design information encoded by the Decorator pattern [15] is specified as follows:

```
decorator(Component, ConcreteComponent, Decorator,
          ConcreteDecoratorSet, Operation,
          AddBehavior, Components) :-
    assert(abstractclass(Component)),
    assert(method(Component, public, Operation)),
    assert(inherit(Component, ConcreteComponent)),
    assert(class(ConcreteComponent)),
    assert(method(ConcreteComponent, public, Operation)),
    assert(inherit(Component, Decorator)),
    assert(abstractclass(Decorator)),
    assert(variable(Decorator, private, Components,
                  Component)),
    assert(method(Decorator, public, Operation)),
    assert(invoke(Decorator, Operation, Components,
                  Operation)),
    forall(member(ConcreteDecorator, ConcreteDecoratorSet),
           assert(inherit(Decorator, ConcreteDecorator))),
    forall(member(ConcreteDecorator, ConcreteDecoratorSet),
           assert(class(ConcreteDecorator))),
    forall(member(ConcreteDecorator, ConcreteDecoratorSet),
           assert(method(ConcreteDecorator, public, Operation))),
    forall(member(ConcreteDecorator, ConcreteDecoratorSet),
           assert(method(ConcreteDecorator, public,
                        AddBehavior))),
    forall(member(ConcreteDecorator, ConcreteDecoratorSet),
           assert(invoke(ConcreteDecorator, Operation,
                        Decorator, Operation))),
    forall(member(ConcreteDecorator, ConcreteDecoratorSet),
           assert(invoke(ConcreteDecorator, Operation,
                        ConcreteDecorator, AddBehavior))).
```

The main goal of the Decorator pattern is to add responsibilities to individual objects dynamically and transparently without affecting other objects. As shown in Figure 2, the decorator forwards requests to the component and may perform additional actions before or after forwarding these requests. This description will be part of the case study presented in the next section. A more comprehensive set of pattern descriptions can be found in [10], where we have provided the Prolog representations of all design patterns in [15].

3 Case Study

In this section, we illustrate the analysis process previously described by checking the properties of components and their compositions within a hypermedia case study. This case

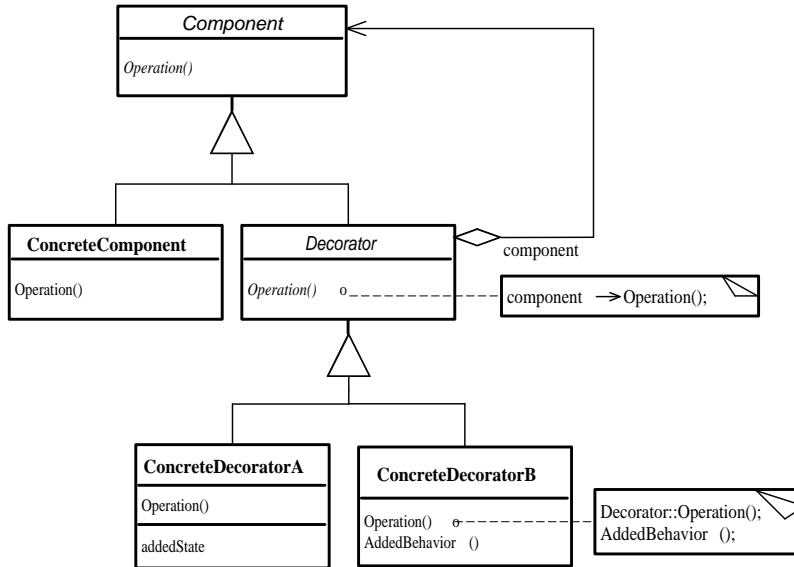


Figure 2: The Decorator Pattern

study is related to the design of the LivePage system [14].

Hypermedia design patterns [28] have been proposed to reuse design experience, to improve communication within and across software development teams, to capture explicitly the design decisions made by designers, and to record design tradeoffs and design alternatives in hypermedia applications. A number of hypermedia design patterns have been discovered, such as the Navigation Observer pattern, the Active Reference pattern, the Navigational Contexts pattern, and the Information on Demand pattern. A comprehensive catalog of hypermedia design patterns can be found in [17]. In what follows, using the Active Reference pattern and the Navigational Context pattern (from [28]) as examples, we show how to describe design pattern components and their compositions, and how to reason about composition properties using a model checker (XMC).

3.1 Two Design Components

In many hypermedia applications, particularly those with spatial or time structures, the user usually needs to have visual knowledge about his/her current location in terms of spatial or time information throughout the navigation

process. This information not only helps the user to find out the current position in complex navigation spaces, but also allows the user to change his/her position freely. The Active Reference pattern [28] 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 the user is navigating a multi-page document. The OMT description of this pattern is shown in Figure 3. The *Component* class is the navigation component, in which the *Show* operation is defined to show its contents on the screen. The *Notify* operation is used to notify the change of the current navigation status, e.g., closing the display of the current component and opening that of another component. The *Reference* class is an abstract class which defines an interface with a list of operations. The *Update* operation is used to change the visual highlight, which shows the current position in the navigation structure, when a new navigation component is displayed. The *Display* operation displays or refreshes the active reference on the screen. The *GoTo* operation is defined to change the current position by directly

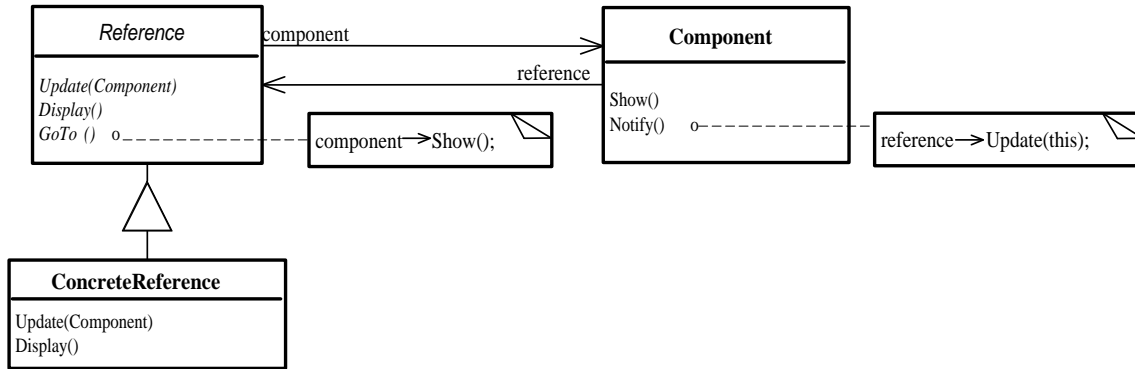


Figure 3: The Active Reference Pattern

selecting an item on the active reference in order to display the corresponding component. The *ConcreteReference* class implements different concrete active references. For instance, an index can be a textual active reference to a document; a map can be a graphic active reference to a travel information system.

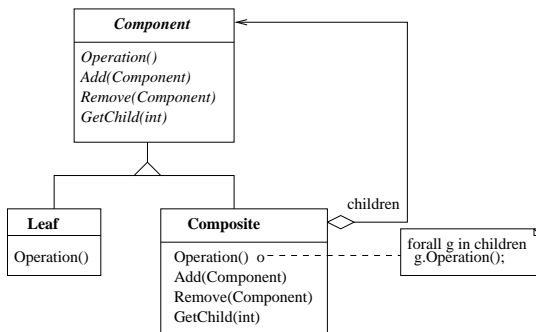


Figure 4: The Composite Pattern

Hypermedia applications usually involve navigating collections of nodes, which may be explored in different ways according to the task the user is performing. For example, collections of paintings may be studied author by author, or explored by different categories, e.g., nature paintings or architecture paintings. The Navigational Contexts pattern separates the context information from the contents of a hypermedia component and dynamically attaches different context information to a component. This enrichment of the navigation interface, when a component is visited in that context, can be achieved by a design solution that is

similar to the Decorator pattern [15] (see the descriptions in Section 2.2). If the collections of hypermedia components are modeled as the aggregate described in the Composite pattern [15] (see the OMT diagram in Figure 4), the Navigational Contexts pattern can be seen as the integration of the Decorator pattern and the Composite pattern. Its OMT diagram is shown in Figure 5. The names of some classes and operations have been changed to represent the corresponding meanings in the Navigational Contexts pattern, e.g., *Decorator* has been replaced by *Context*, *Leaf* has been replaced by *Content*, and *Operation* has been replaced by *Show*. An example of the application of the Navigational Contexts pattern can be found in [8]. In this case, all the blue buttons constitute context information that is dynamically appended to the corresponding page content by the LivePage system [14].

3.2 Representation

The representation of the Active Reference pattern in XSB Prolog follows:

```
active_reference(Component, Reference,
                ConcreteReferenceSet, Show,
                Notify, Update, Display, GoTo) :-

    assert(class(Component)),
    assert(method(Component, public, Show)),
    assert(method(Component, public, Notify)),
    assert(variable(Component, private, reference,
                    Reference)),
    assert(inline(Component, Notify, reference, this)),

    assert(abstractclass(Reference)),
    assert(method(Reference, public, Update, void,
                  component, Component)),
    assert(method(Reference, public, Display)),
```

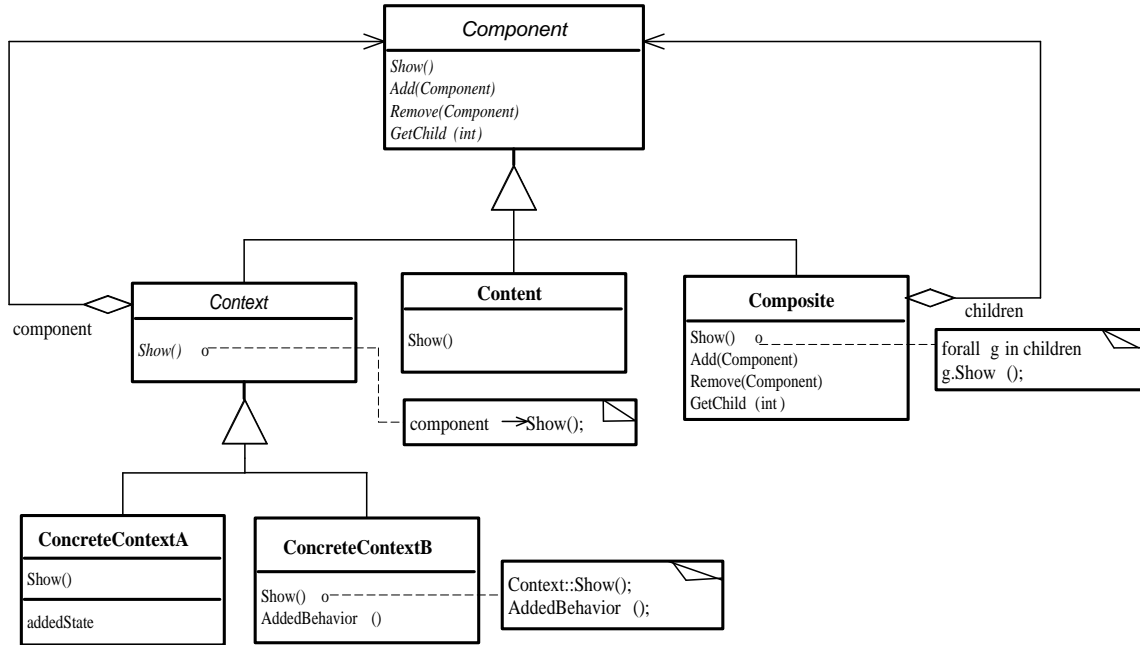


Figure 5: The Navigational Contexts Pattern

```

assert(method(Reference, public, GoTo)),
assert(invoke(Reference, GoTo, component, Show)),

forall(member(ConcreteReference, ConcreteReferenceSet),
  assert(class(ConcreteReference))),
forall(member(ConcreteReference, ConcreteReferenceSet),
  assert(method(ConcreteReference, public, Update,
    void, component, Component))),
forall(member(ConcreteReference, ConcreteReferenceSet),
  assert(method(ConcreteReference, public, Display))).

```

The behavioral aspect of the Active Reference pattern is modeled in terms of the collaboration among the methods of each class. This behavior is represented in XL, where each XL process describes the behavior of a method. The `ref` process defines the behavior of this pattern as the parallel composition of these processes:

```

refGoTo(Reference, GoTo, Component, Show, C) ::=
  out(C, (Reference, GoTo, Component, Show))
  o action(ref_goTo(Reference, GoTo)).

componentShow(Reference, GoTo, Component, Show, C) ::=
  in(C, (Reference, GoTo, Component, Show))
  o action(component.Show(Component, Show)).

componentNotify(Component, Notify, Reference,
  Update, R) ::=
  out(R, (Component, Notify, Reference, Update))
  o action(component.Notify(Component, Notify)).

refUpdate(Component, Notify, Reference, Update, R) ::=
  in(R, (Component, Notify, Reference, Update))
  o action(ref_Update(Reference, Update)).

```

```

ref(Reference, Component, GoTo, Update, Show, Notify) ::=
  refGoTo(Reference, GoTo, Component, Show, C)
  | componentShow(Reference, GoTo, Component, Show, C)
  | componentNotify(Component, Notify, Reference,
    Update, R)
  | refUpdate(Component, Notify, Reference, Update, R).

```

The Prolog description of the Navigational Contexts pattern can be achieved easily by incorporating all corresponding name changes (discussed in the previous section) and instantiating the Prolog descriptions of the *Decorator* pattern² and the *Composite* pattern³ as follows:

```

navigational_contexts(Component, Context,
  ConcreteContextSet, Content, Composite,
  Show, Add, Remove, GetChild, AddedBehavior,
  Components, Children) :-
  decorator(Component, Content, Context,
    ConcreteContextSet, Show, AddedBehavior, Components),
  composite(Component, Composite, Content, Children, Show).

```

The behavioral aspect of the Navigational Contexts pattern is also modeled in terms of the collaboration among the methods of each class in XL as follows:

²The Prolog description of the Decorator pattern was shown in Section 2.2.

³The Prolog description of the Composite pattern can be found in [10].

```

contextShow(Context, Component, Concrete, Show,
             R, T) ::=
    in(R, (Concrete, Show, Context, Show))
    o out(T, (Context, Show, Component, Show))
    o action(context_Show(Context, Show)).

compShow(Context, Component, Concrete, Content,
          Composite, Show, P, T) ::=
    { in(P, (Composite, Show, Component, Show))
      # in(T, (Context, Show, Component, Show)) }
    o
    { contextShow(Context, Component, Concrete, Show)
      # contentShow(Content, Show)
      # compositeShow(Composite, Component, Show)
    }
    o compShow(Context, Component, Concrete, Content,
                Composite, Show).

contentShow(Content, Show) ::=
    action(content_Show(Content, Show)).

compositeShow(Composite, Component, Show, P) ::=
    out(P, (Composite, Show, Component, Show))
    o action(composite_Show(Composite, Show)).

concreteShow(Concrete, Context, Show, R) ::=
    out(R, (Concrete, Show, Context, Show))
    o action(concrete_Show(Concrete, Show))
    o action(AddedBehavior).

nav(Context, Component, Concrete, Content, Composite,
     Show) ::=
    contextShow(Context, Component, Concrete, Show,
                R, T)
    | compShow(Context, Component, Concrete, Content,
                Composite, Show, P, T)
    | contentShow(Content, Show)
    | compositeShow(Composite, Component, Show, P)
    | concreteShow(Concrete, Context, Show, R).

```

3.3 Instantiation

In the previous section, we have presented the generic description of each pattern in XL. To use a pattern in a specific application, we need to instantiate it to include application domain information. This process can be easily achieved by unifying the arguments of the description of each design pattern component with terms representing domain information. For instance, the Active Reference pattern can be instantiated as the design of a collection of paintings in a museum that uses a map with an active reference to the current visiting location. The current location is highlighted on the map. This instantiation process is represented as follows: *active_reference(painting, ref_interface, [map], show, notify, display, goTo)*. The behavioral aspect can be instantiated as follows: *ref(ref_interface, painting, goTo, update, show, notify)*. In this way, the design information related to the Active Reference pattern is added to the XSB Prolog database and can be composed with the design information related to

other instantiated design components.

For the example of exploring the paintings in a museum, the user may need to study them in different contexts. This design decision can be recorded by the application of the Navigational Contexts pattern as follows: *navigational_context(painting, context, [text, button], content, composite, show, add, remove, getChild, showButtons)*. In addition, the behavioral aspect can be instantiated as follows: *nav(context, [text, button], painting, content, composite, show)*. There are two kinds of context information attached to the content in this design: one is a list of buttons, e.g., first, next, previous, last buttons, which are used to connect all hypermedia components with similar context by a linked list to make the navigation process easier; the other is related to the content of the corresponding hypermedia component. For example, the Van Gogh's painting *Sun Flowers* can be reached while exploring paintings about nature, where the first kind of context information is related to the list of buttons connecting to the next, previous, first, or last paintings about nature in this collection, and the second kind of context is related to the information about the natural aspects of the painting. On the other hand, *Sun Flowers* can be accessed as a Van Gogh's work, where the first kind of context information can be the list of buttons connecting to the next, previous, first, or the last Van Gogh's painting, and the second kind of context contains information about the relationship of this painting with his other paintings.

3.4 Integration

As the application requires an active map showing the current position of the user in a museum and requires the user to be able to explore the museum according to different contexts, we can combine two design pattern instances to achieve this goal.

The composition of the two design components can be carried out by overlapping their common part as, for example, it is shown in Figure 6, where the *painting* class is the overlapping part of the two design components (see Figure 3 and Figure 5). The integration, which is the model of the design composition, can be

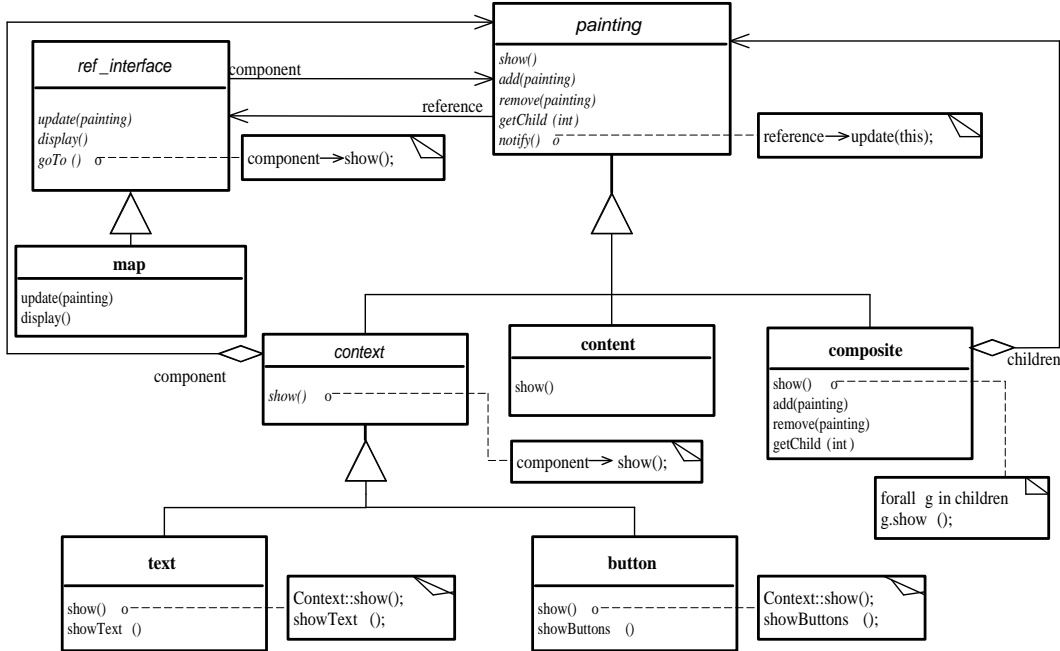


Figure 6: The Design Composition

described as a process in XL:

```
reference_context ::=
  active_reference(painting, ref_interface,
    [map], show, notify, display, goTo)
  | navigational_context(painting, context,
    [text, button], content, composite,
    show, add, remove, getChild, showButtons)
  | ref(ref_interface, painting, goTo, update,
    show, notify)
  | nav(context, [text, button], painting,
    content, composite, show).
```

3.5 Design Analysis Results

The goal of the design analysis is to find design composition errors by verifying properties about the combination of the design components through model checking techniques. We should notice that composition errors are hard to detect by visual inspection of the designs. In this section, we describe the discovery and correction of two design errors in the resulting design composition: one error is related to the structural aspect and the other is related to the behavioral aspect of the design composition.

The first error of this design composition we found by using the XMC/XSB tool was that the *painting* class was defined as both an abstract class and a concrete class in the composition of the Prolog descriptions. The reason for

this inconsistency is 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 can not be overlapped with the *Component* class in the Navigational Contexts pattern and instantiated as the *painting* class. Instead, it should overlap with the concrete Component (the *content* class) in the Navigational Contexts pattern. Therefore, the *ref_interface* class in Figure 6 (an instance of the *Reference* class in the Active Reference pattern in Figure 3) should have an association relationship with the *content* class instead of having an association relationship with the *painting* class.

As we continued analyzing the design composition, we found another error in the resulting design. The idea of the Active Reference pattern is to have a permanent and visible reference to a navigation structure and to be able to change the current position by calling the *goTo* operation in the *ref_interface* class. Therefore,

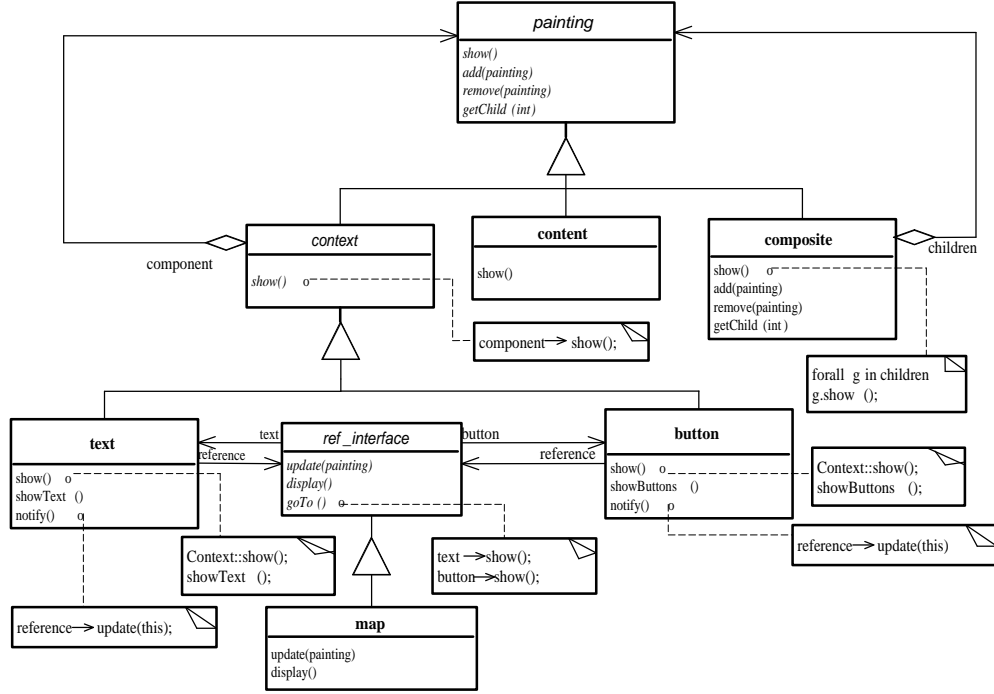


Figure 7: The Modified Design Composition

the invocation of the *goTo* operation should eventually invoke both the *show* operation in the *content* class to display the content of a hypermedia component, and the *show* operations in the concrete Context classes (i.e., the *text* class and the *button* class) to display the context information of the hypermedia component. These liveness properties are described generically in μ -calculus as follows:

```

% Content::Show will be eventually invoked
% after Reference::GoTo is invoked
liveness1(Reference, GoTo, Content, Show) ==
  [ref_GoTo(Reference, GoTo)] formula1(Content, Show)
  /\ [-] liveness1(Reference, GoTo, Content, Show).

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

form1(Content, Show) +=
  <content_Show(Content, Show)> tt
  \/ [-ref_GoTo(Reference, GoTo)] form1(Content, Show).

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

formula2(ConcreteContext, Show) +=
  <concrete_Show(ConcreteContext, Show)> tt

```

```

  \/ form2(ConcreteContext, Show)
  \/ [-] formula2(ConcreteContext, Show).

form2(ConcreteContext, Show) +=
  <concrete_Show(ConcreteContext, Show)> tt
  \/ [-ref_GoTo(Reference, GoTo)] form2(ConcreteContext, Show).

```

They can be instantiated to represent the liveness properties in this application as follows:

```

liveness1(ref_interface, goTo, content, show).
liveness2(ref_interface, goTo, text, show).
liveness2(ref_interface, goTo, button, show).

```

The model checking results of these liveness properties indicate that the first liveness property, that the *show* operation in the *content* class is eventually invoked, holds. However, the second liveness property, that the *show* operation in the concrete Context class (i.e. the *text* class and the *button* class) is eventually invoked, does not hold. Therefore, when the user clicks on the active reference (e.g., the map of a museum) to change the current position, only the content of the newly chosen component will be displayed. The context information (such as the buttons) of this component will not be shown; in this case, we have lost all context

information and are not able to navigate properly using the context links. The solution to this problem is to move the overlapping part further down to the concrete Context classes as shown in Figure 7. After making this change, we were able to verify using the model checker that both liveness properties hold.

4 Discussions

Formal methods have been successfully used in many safety-critical systems, such as spacecraft, airplane, medical systems, and nuclear plant systems. Failures in the execution of these systems may result in loss of people's lives and/or large amounts of money as, for example, in the case of the failed launch of the \$500 million Ariane 5 in 1996 [19] and in the case of the failure of the London Ambulance Service [9]. Therefore the cost of using formal methods to develop these systems is justified.

Besides being applied in safety-critical systems, formal methods are used more and more in commercial software system applications. Software users become frustrated when software crashes from time to time and, as a result, there has been an increasing demand for high quality software systems. The usage of formal methods is one way to improve software quality. There are many experiments on applications of formal methods in software development such as the ones described in [30, 13, 34].

Widmaier et al. [33] conducted an experiment about the development of a software system by two independent groups. One group used formal methods and the other group used the traditional waterfall model. Even with the time-to-market requirement in mind, it was discovered that the group that used formal methods produced a more reliable software system than the group that did not use formal techniques. In this case study, the authors also indicated that the group that did not use formal methods depended on one expert developer to build their system on time. This might be undesirable from both the management and the engineering points of view because depending on an experienced developer may involve more risks than relying on good methodologies.

Using formal methods with tool support

might also make it easier to find errors in software development. In [1], we have shown that, by using a lightweight formal method, we could find a subtle design error which would be otherwise difficult to detect only by a visual inspection of the informal design documentation. In addition, according to Barry Boehm, most subtle errors are far more expensive to detect than to correct. We believe that formal methods can help to detect these errors.

Hissam [18] described a case study showing how a design error could be found in the maintenance phase. He concluded that a deep knowledge about different systems and significant debugging skills were needed in order to be able to find errors in the maintenance phase, especially when the software system is based on components.

5 Related Work

Keller et al. [20] described a methodical approach to design composition that was illustrated as a process within a four-dimensional design space. Although our approach is also in the area of software composition, it focuses on the formal, declarative, transformational and property-based aspects of design composition.

Pal [25] investigated the law-governed support to realize design patterns. He defined some rules and constraints of design patterns. However, in his work the property checking is performed at the implementation level. He did not discuss the interactions among the different design patterns that were integrated.

Batory [6] provided domain-independent algorithms to validate component compositions using the GenVoca model of software generators. In addition to syntactical checking, such as type checking, they provided design rule (domain-specific constraint) checking to ensure correctness. The design rule checking was achieved through the debugging capabilities of a general tool based on attribute grammars. In contrast, our work focuses on reasoning about design compositions.

Riehle [27] proposed an analysis method for the composition of design patterns. Role diagrams were introduced to describe the patterns and a role relation matrix was used to visually

depict the composition constraints. His work was restricted to deal with the patterns based on object collaborations, and lacked more general pattern descriptions and a formal treatment to deal with the correctness of the pattern combinations.

The formalization of design and architecture patterns has been proposed in [2, 23]. Although Mikkonen [23] has discussed the composition of two design patterns based on a formal method, his approach relies on a specific specification language (DisCo). In this case, composition correctness is defined in terms of refinements in DisCo. However, our approach focuses on the specification of design components and their composition and on using model-checking techniques to verify composition properties. Another difference between his approach and ours is that his approach focuses on formalizing design patterns, whereas our work deals with a more general approach based on design components [20].

6 Conclusions

Errors in software designs are very difficult to be detected and expensive to correct if propagated to the implementation phase because these errors may be hidden in complex implementation structures and sometimes in thousands of lines of code. Using an analogy, if detecting a design error at the design level can be considered as looking for a button in a swimming pool, then detecting the same error at the implementation level may be seen as looking for the button in a lake.

In this paper we describe an approach based on model checking to analyze the properties of components and their compositions at the design level. This approach enables designers to perform relevant analysis activities by allowing them to verify properties about the integration of design components. We illustrate our analysis techniques through a case study about the integration of hypermedia design components. However, our approach is not restricted to the hypermedia domain. We have analyzed the design of a general system sort [1] and developed a method to prove structural and behavioral correctness of generic design compositions [11].

Our approach has several advantages: first, it allows us to find errors about the design composition early in the development process and save cost related to correcting them later. Second, it provides mechanisms to achieve the automated verification of software design properties. Third, it promotes reuse since we assume that the generic design component representations can be stored in a component repository and retrieved later to be instantiated and integrated in an application. Fourth, the hypermedia case study indicates that there is a potential for practical benefit in the application of formal methods in hypermedia design. Furthermore, as the composition of components can be treated as a single component, we believe that our design analysis can scale up incrementally in order to address integration properties of large component-based software systems.

Finally, our approach showed that the structural and behavioral aspects of design components can be described in the same formal language and, in this way, we can analyze the properties of both of these aspects of design composition using our techniques. This possibility may also lead to interesting and relevant design analyses such as the analysis of the interactions between the structural and the behavioral of a software design.

Acknowledgements

The author would like to thank IBM Canada for providing a CAS Fellowship to support this work. This work is also partially supported by NSERC and CSER.

The work would not have been possible without the helpful suggestions and guidance of the author's supervisor Donald D. Cowan and Paulo S.C. Alencar. The author would also like to thank all members of the University of Waterloo software engineering research community and the anonymous referees for their useful comments to improve this paper.

About the Author

Jing Dong is a Ph.D. candidate in the Department of Computer Science at the University of Waterloo. He received a M.Math de-

gree in computer science from the same department in 1997. His primary research interests include component-based software engineering, formal methods, design composition, component-based document design, design pattern, and verification and validation.

References

- [1] Paulo Alencar, Donald Cowan, Jing Dong, and Carlos Lucena. A Pattern-Based Approach to Structural Design Composition. *Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference (COMPSAC), Phoenix USA*, pages 160–165, October 1999.
- [2] P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena. A Formal Approach to Architectural Design Patterns. *Proceedings of the Third International Symposium of Formal Methods Europe*, pages 576–594, 1996.
- [3] Joanne M. Atlee and John Gannon. State-Based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.
- [4] Geoff Barrett. Model Checking in Practice: The T9000 Virtual Channel Processor. *IEEE Transactions on Software Engineering*, 21(2):69–78, November 1998.
- [5] Samik Basu, Scott A. Smolka, and Orson 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.
- [6] D. Batory and B.J. Geraci. Validating Component Composition in Software System Generators. *Proceedings of the 4th International Conference on Software Reuse*, pages 72–81, April 1996.
- [7] M.C. Browne, E.M. Clarke, and D.L. Dill. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Transactions on Computer*, C-35(12):1035–1044, Dec. 1986.
- [8] Donald D. Cowan. Personal Homepage. <http://csg.uwaterloo.ca/~dcowan>.
- [9] Darren Dalcher. Disaster in London: The LAS Case Study. *Proceedings of the 6th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems(ECBS)*, pages 41–52, April 1999.
- [10] Jing Dong. A Transformational Process-Based Approach to Object-Oriented Design. *Master's Thesis, Computer Science Department, University of Waterloo*, 1997.
- [11] Jing Dong, Paulo Alencar, and Donald 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.
- [12] E. Emerson and E.M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2:241–266, 1982.
- [13] Martin S. Feather. Rapid Application of Lightweight Formal Methods for Consistency Analyses. *IEEE Transactions on Software Engineering*, 24(11):949–959, November 1998.
- [14] B. Fraser, J. Roberts, G. Pianosi, P. Alencar, D. Cowan, D. Germán, and L. Nova. Dynamic Views of SGML Tagged Documents. *Proceedings of the ACM SIGDOC*, pages 93–98, Sept. 1999.
- [15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.
- [16] David Garlan, Robert Allen, and John 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.

- [17] Daniel M. Germán and Donald D. Cowan. Towards a Unified Catalog of Hypermedia Design Patterns. *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, Jan. 2000.
- [18] Scott 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, Nov. 1998.
- [19] J.-M. Jezequel and B. Meyer. Design by Contract: The Lessons of Ariane. *IEEE Computer*, pages 129–130, Jan. 1997.
- [20] Rudolf K. Keller and Reinhard Schauer. Design Components: Towards Software Composition at the Design Level. *Proceedings of the 20th International Conference on Software Engineering*, pages 302–311, 1998.
- [21] D. Kozen. Results on the Propositional μ -calculus. *Theoretical Computer Science*, 27:333–354, 1983.
- [22] Christian Krämer and Lutz Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. *Proceedings of the Working Conference on Reverse Engineering, IEEE CS press, Monterey, Nov. 1996*.
- [23] Tommi Mikkonen. Formalizing Design Pattern. *Proceedings of the 20th International Conference on Software Engineering*, pages 115–124, 1998.
- [24] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
- [25] Partha pratim Pal. Law-Governed Support for Realizing Design Patterns. *Technology of Object-Oriented Languages and Systems (TOOLS), USA*, pages 25–34, July 1995.
- [26] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient Model Checking Using Tabled Resolution. *Proceedings of the 9th International Conference on Computer Aided Verification (CAV), Haifa Israel, LNCS1243, Springer-Verlag, July 1997*.
- [27] Dirk Riehle. Composite Design Patterns. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA), USA*, pages 218–228, October 1997.
- [28] Gustavo Rossi, Daniel Schwabe, and Alejandra Garrido. Design Reuse in Hypermedia Applications Development. *Proceedings of the ACM International Conference on Hypertext*, pages 57–66, April 1997.
- [29] Colin Stirling. An Introduction to Modal and Temporal Logics for CCS. *Lecture Notes in Computer Science 491, Springer-Verlag*, pages 1–20, 1991.
- [30] Kevin J. Sullivan, John Socha, and Mark Marchukov. Using Formal Method to Reason about Architectural Standards. *Proceedings of the 19th International Conference on Software Engineering*, pages 503–513, May 1997.
- [31] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Longman, Reading, Mass., 1998.
- [32] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.
- [33] James C. Widmaier, Carol Smidts, and Xin Huang. Producing More Reliable Software: Mature Software Engineering Process vs. State-of-the-Art Technology? *Proceedings of the 22nd International Conference on Software Engineering*, pages 88–93, June 2000.
- [34] Jeannette M. Wing and Mandana Vaziri-Farahani. A Case Study in Model Checking Software Systems. *Science of Computer Programming*, 28:273–299, 1996.
- [35] XSB. The XSB Logic Programming System, Version 2.1. Available from <http://www.cs.sunysb.edu/~sbprolog>, 1999.