CS 2001 -13

Process-Based Representation and Analysis of Framework Instantiation

Paulo S. C. Alencar, Donald D. Cowan
Computer Science Department, University of Waterloo – Waterloo, Canada
Toacy C. Oliveira, Carlos J.P. Lucena
Department of Computer Science, PUC-Rio – Rio de Janeiro, Brazil

***Abstract*—Object-oriented frameworks are currently regarded as a promising technology for reusing designs and implementations. However, developers find there is still a steep learning curve when extracting the design rationale and understanding the framework documentation during framework instantiation. Thus, instantiation is a costly process in terms of time, people and other resources. These problems raise a number of questions including: "How can frameworks be instantiated more quickly and with greater ease? How can the same high-level design abstractions that were used to develop the framework be used during framework instantiation instead of using source code as is done currently? How can we capture the designers' knowledge of the framework in order to compensate for the loss of key development personnel? How can we raise the level of abstraction in which the framework evolution and instantiation is expressed, reasoned about and implemented?" In this paper we present a process-based approach to framework instantiation that addresses these issues. Our main goal is to represent the framework architectural design models in an explicit and declarative way, and support changes to this architecture based on explicit instantiation processes and activities while maintaining system integrity, invariants, and general constraints. In this way, the framework instantiation and evolution can be performed in a valid and controlled way.**

To accomplish our goal, we introduce a process-oriented description of framework instantiation as well as a formal specification of these processes that allows us to reason about instantiation using model checking techniques. Discovering instantiation errors and analyzing alternative architectures and designs early in the development process could certainly lower the cost and effort in fixing them. Various forms of analysis can be performed using our approach to check properties such as: structural evolution properties, pre- and post-conditions of an instantiation, the validity of the processes associated with extension points, the order of the instantiation processes, the safety and liveness of these processes, reachability, and deadlocks. We illustrate our approach with DrawingTool, a framework to provide drawing features of a case tool.

***Index terms*-- frameworks, instantiation, software process, software architecture, design, specification, automated verification, Prolog, model checking, design analysis.**

I. Introduction

It is widely believed that reuse of software designs and implementations leads to substantial gains in productivity. Reuse proposes the re-application of structured knowledge to achieve qualitative and quantitative improvements during the software development process. In the reuse scenario, object-oriented techniques such as frameworks [1,2] are currently regarded as a promising technology for reusing designs and implementations because of the amount of knowledge encapsulated within a component or a software application.

During the framework instantiation or reuse process, the re-user [3] uses the framework specification as an initial/partial design of the application being developed. The developer reusing the framework adapts the design to incorporate application-specific requirements.

Framework Instantiation Problems. However, developers find there is still a steep learning curve when extracting the design rationale and understanding the framework documentation during framework instantiation. Thus, instantiation is a costly process in terms of time, people and other resources.

The framework specifications are usually unstructured and use natural language to describe the artifacts. The lack of clear, detailed and complete documentation leads to a time-consuming instantiation process, which negates one of the most valuable properties of reuse, that is, a significant reduction in application development time. In fact achieving high productivity with framework reuse can take months [2].

Another problem relates to the consistency of the final design. Some instantiation processes introduce unexpected states that can violate some of the framework design constraints. Therefore, it is important that the framework documentation provides a set of properties that must be preserved after the each instantiation process is performed.

Questions. These problems raise a number of questions including: "How can frameworks be instantiated more quickly and with greater ease? How can the same high-level design abstractions that were used to develop the framework be used during framework instantiation instead of using source code as is done currently? How can we capture the designer's knowledge of the framework in order to compensate for the loss of key development personnel? How can we raise the level of abstraction in which the framework evolution and instantiation is expressed, reasoned about and implemented?"

Goal. In this paper, we introduce a process-based approach that addresses these issues. We represent framework instantiations through processes that indicate "how" the instantiation should be accomplished, and "what" instantiation processes need to be used to for a certain instantiation activity. This representation also supports analysis. Our main goal is to support an explicit declarative representation of the framework architectural design models, and allow changes to this architecture based on explicit instantiation processes and activities. Further, these processes and activities will no compromise the system integrity, invariants, and general constraints.

Approach. To accomplish our goal, we have defined a generic structural UML-based specification formalism for frameworks and their extensions and created techniques through which the instantiation of such extensions can be verified. Framework instantiations are based on the notion of extension points, or explicit 'places' in the framework design where the framework can be extended. We use the UML notation extension mechanisms to represent those extension points and other concepts relevant to the framework instantiation such as the kind of instantiation.

The process-oriented descriptions of the framework instantiation as well as formal specification of these processes allow us to reason about these instantiations using model checking techniques. In this way, we will be able to analyze alternative architectures and designs that could be applied in the instantiation process. Discovering instantiation errors and analyzing alternative architectures and designs early in the development process could certainly lower the cost and effort in fixing them. Various forms of analysis can be performed using our approach to check properties such as: structural evolution properties, the pre- and post-conditions of the instantiations, the validity of the processes associated with the extension points, the order of the instantiation processes, the safety and liveness of these processes, reachability, and deadlocks. We illustrate our approach with DrawingTool, a framework to provide drawing features of a case tool.

Contributions. The main contributions of this paper can be summarized as follows: (i) generic techniques to represent and specify frameworks and their extension points. This generic specification can be used to instantiate a concrete framework instantiation using application domain knowledge. The specification language is based on the declarative programming language, Prolog; (ii) a process-based approach to specifying the instantiation processes, and the constraints that should hold when the framework is instantiated using these processes; (iii) techniques to supports the verification of the framework architecture and the instantiation processes in order to analyze through Prolog proofs and model checking whether instantiation properties hold; (iv) a case study applying the systematic approach to specify and verify the frame work extension of DrawingTool, a framework that provides the drawing features of a case tool.

Paper Outline. The remainder of this paper is organized as follows. Section 2 describes the gist of our analysis techniques including an approach to framework instantiation analysis based on abstract design specifications and model checking. Section 3 gives details of a case study illustrating the framework instantiation analysis approach. We analyze extensions to a framework called DrawingTool. The last sections focus on related work and conclusions.

II. A PROCESS-BASED APPROACH

In this section, we introduce our process-based approach to framework instantiation including techniques for the representation, instantiation, and analysis (property checking) of frameworks and their extensions. We also describe the model checking technique used to analyze framework instantiations to ensure that framework evolution is valid.

A. Overview

An overview of our process-based approach is shown in Figure 1 using the re-user's point of view. We try to hide formal representations in order to provide a common level of abstraction between the design and the instantiation processes. In this view we have the concept of a reusable artifact that is composed of three parts.

The first part is a set of decorated class diagrams, which are represented in UML [4] graphical notation and are used to specify the framework design. Note that we use an extended version of UML in order to represent the frameworks extension points [5].
The second part is a Cookbook, in the same sense of [6]. This cookbook will guide the instantiation process and it is expressed in a specially defined language.
The last part of the artifact is a set of properties that need to be checked in order to validate the instantiation process.

This reusable artifact is then introduced into the tool to begin the instantiation process. During this process, the re-user is asked to provide class names and choice conditions.

Once the process is finished the re-user has the final application design.

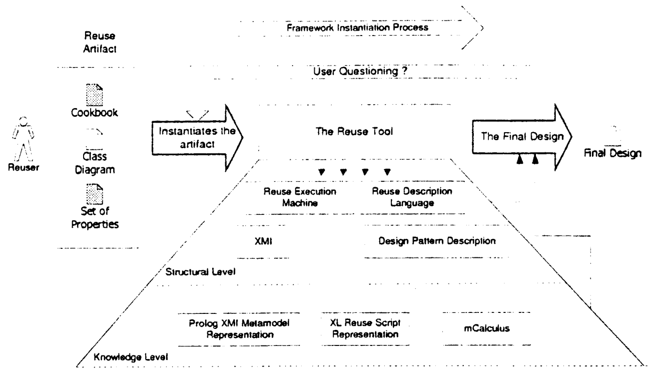


Figure 1

It is also important to note that internally the tool is divided into two layers (Figure 1). The first layer is the structural layer and is responsible for the structural modification in the design, which means, responsible for generating a user friendly output for the instantiation process. This can be accomplished with the use of the XMI OMG standard.

The other layer is responsible for capturing the knowledge behind the process. To achieve this the structural of each framework element is translated into Prolog representations. The instantiation processes are represented in a process-based instantiation language and later translated into process algebra representations. We also analyze whether the instantiation processes obey evolution properties described in temporal logic that are checked against the framework model. Since graphical notations are more intuitive and generally easier to understand, we use such a notation as the method for designers and developers to build their designs. In this way, they are not directly exposed to the intricacies of the underlying formalisms. On the other hand, the informal graphical models, which lack precision and may be ambiguous, cannot be the base for a formal analysis. A formal model allows us to reason about the properties of the design compositions and verifies our designs.

B. *Formal Specification and Verification Techniques*

Besides presenting language constructs for the representation of the instantiation process, formal specification and (automated) verification techniques constitute essential tools in our representation and analysis of framework instantiations. For the analysis, the framework structural UML-based models and properties are specified in first-order logic and checked using Prolog [7]. The framework instantiation processes expressed in our instantiation language are translated to XL (a process language) and the properties related to the framework instantiation process are specified in μ-calculus and verified through model checking techniques [8,9,10,11].

Model checking entails comparing two formal objects (Σ, ϕ) Σ the software design components and their compositions modeled in logic, and ϕ the properties of these components represented as logic formulas. One assumes that if a formula ϕ is true in the model Σ, then the corresponding property holds in the model of the design. We use a model checker as a black box 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. When a property violation is found, we can go back to check and update the design composition. One reason why this verification technique is so promising is that model checking can be automated for many temporal logics.

To automate the analysis of the framework instantiations, we use XMC [8,12], a model checker for verifying temporal properties of a software system. XMC is written in the XSB table Prolog programming system [7]. Temporal properties are expressed in the alternation-free fragment of the μ-calculus [8,13], a very expressive temporal logic; the system to be verified is described in the model specification language for XMC (called XL) which is a highly expressive extension of value-passing CCS [14]. Prolog terms and predicates are used to represent values and computations. Thus specifications can make use of recursive data structures and computations. XMC has been successfully used to verify various systems as documented in [12].

In the next subsections we describe each phase of our design analysis approach. In particular, we show how the structure is specified in XSB Prolog by rules and facts and the instantiation processes are represented in a framework instantiation language and described in XL, the model specification language for XMC. In this way, we will be able to verify structural properties using the XSB Prolog deductive facilities and verify process-based framework instantiations properties expressed in the μ-calculus temporal logic using the XMC model checker.

C. *Representation*

In the initial phase, the framework designs are represented in Prolog and stored in an XSB Prolog database. There are several advantages of using XSB Prolog as a repository of design knowledge. First, the representations of these components can be reused by instantiating the corresponding generic Prolog descriptions of each design element when it is applied to produce a concrete domain-specific representations. Second, the properties and constraints of each design element can be described and proved in Prolog. Third, the addition and removal of structural facts about design elements can be carried out using the Prolog *assert* and *retract* clauses. Finally, design elements can be recovered through Prolog deductive facilities.

Design Representation Primitives. The structural UML-based representation of the framework elements is specified in XSB Prolog 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, or inheritance. The *argument* part contains generic information about an entity or a relation such as the information on the participants in an inheritance relation. In the following, we present the syntax and the meaning of the design primitives used in this paper[1]:

- *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, P, ,T_1, P_2 ,T_2 ,...)*: 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, C_f ,O, O_f, 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.
- *element(E_1 , S_1 , E_2 , S_2 , ...)*: E_1 is an element of set S_1. E_2 is an element of set S_2, and so on.

Example. As an example, suppose a framework has a Decorator pattern [17] as one of its design elements. The main goal of the Decorator pattern is to add responsibilities to individual objects dynamically and transparently without affecting other objects. As shown in [17,15,16], the decorator forwards requests to the component and may perform additional actions before or after forwarding. The *ConcreteComponent* defines an object to which additional responsibilities can be attached, while the *ConcreteDecoratorA* and *ConcreteDecoratorB* add responsibilities to the component. The XSB Prolog representation of the design information encoded by the Decorator pattern is shown next:

```
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, ddBehavior))),
forall(member(ConcreteDecorator,
ConcreteDecoratorSet),
  assert(invoke(ConcreteDecorator, Operation, Decorator,
            Operation))).
forall(member(ConcreteDecorator,
ConcreteDecoratorSet),
  assert(invoke(ConcreteDecorator, Operation,
            ConcreteDecorator, AddBehavior))).
```

The Prolog rule, *decorator*, represents the structural aspect of the Decorator pattern. The arguments of *decorator* denote the generic elements such as classes, attributes, or methods. For example, *Component* and *Decorator* are abstract classes; *Operation* and *AddBehavior* are methods; *Components* represents an object reference. The Prolog operators, *assert* and *retract*, are used to insert or remove certain facts into or from the Prolog database, respectively. The *forall* predicate represents the universal quantification operator. When it is used with the *member* predicate, it can quantify over a set of class names and apply a Prolog rule on selected members.

D. Framework Instantiation Processes

Our process-based instantiation assumes that the framework design contains a set of explicit extension points. An instantiation point is a 'place' in the framework design where the framework can be instantiated. Each extension point has an associated cookbook 'recipe' and can be represented by a tuple (RN, FE, CS, PD), where RN is the recipe name associated with the extension point, FE is the framework element such as class or method to which the extension is attached, CS is the set of constraints associated with the instantiation, and PD is a process description of the tasks and activities that should be performed when instantiation occurs. The set of the extension point

[1] A more comprehensive set of design primitives is described in [15,16].

specifications defines the design space of declared framework instantiations.

Framework Instantiation Processes. We have defined a set of language constructs that can be used for the representation of framework instantiation processes.

COOKBOOK ::=	**cookbook** NAME IP_RECIPE+
IP_RECIPE ::=	IP_REC_LINE+
IP_REC_LINE ::=	[// STRING_EXP] [ELEMENT =] IP_CMD ;
IP_CMD	IP_EXP | **loop** IP_RECIPE **end_loop**
IP_EXP ::=	IP | IP # IP | IP o IP | IP // IP
IP ::=	IP_BASIC [REQUIRES_EXP*] | IP_EXP [REQUIRES_EXP*]
IP_BASIC ::=	IP_CLASS | IP_METHOD | IP_ATTRIBUTE | IP_ELEMENT
IP_CLASS ::=	**class_extension** (CLASS_EXP) | **selection_class_extension** (CLASS_EXP) | **pattern_class_extension** (CLASS_EXP , NAME,LIST)
IP_METHOD ::=	**method_extension** (CLASS_EXP, CLASS_EXP, METHOD_EXP) | **pattern_method_extension** (CLASS_EXP, CLASS_EXP, METHOD_EXP, NAME, LIST)
IP_ATTRIBUTE ::=	**value_selection** (CLASS_EXP, ATTRIB_EXP, LIST) | **value_assignment** (CLASS_EXP, ATTRIB_EXP)
IP_ELEMENT::=	**element_choice** (ELEMENT)
ELEMENT::=	CLASS_EXP | METHOD_EXP | ATTRIB_EXP
CLASS_EXP ::=	CLASS
CLASS::=	STRING_EXP
METHOD_EXP ::=	METHOD
METHOD::=	STRING_EXP
ATTRIB_EXP ::=	ATTRIB
ATTRIB::=	STRING_EXP
LIST::=	(LIST_EXP)
LIST_EXP::=	STRING_EXP , LIST_EXP | STRING_EXP
NAME::=	STRING_EXP
REQUIRES_EXP::=	**requires** ORDER_EXP | **requires** ELEMENT
ORDER_EXP::=	**before** IP | **after** IP | **sync** IP | **exclusive** IP
STRING_EXP::=	**String**

Basic Instantiation Processes. The previous process definitions were based on a number of basic instantiation processes such as **class_extension** and **method_extension**. These processes, which are related to class, methods, attributes, and elements, respectively, are defined next in terms of their functionality:

- Class elements – to modify application classes
  - **class_extension**(C) – to create a subclass of the associated class
  - **select_class_extension**(C) – to select a subclass of the associated class
  - **pattern_class_extension**(N, List) – to create a subclass through pattern application
- Method Element – to modify application methods
  - **method_extension**(M) – to create a method
  - **pattern_method_extension**(M, C, List) – to instantiate a method using a pattern
- Attribute Element – to modify attributes
  - **value_selection**(V, A) – to select a value
  - **value_assignment**(V,A) – to assign a value
- Element – to modify elements
  - **element_choice**(E) – selects an element

E. Framework Instantiation Analysis

Formal Process Specification. The framework process-based instantiations based on the constructs described in the previous sub-section are translated to XL, the model specification language for XMC. XL is a highly expressive extension of value-passing CCS. The syntax of the XL specification is given next:

```
Pdef -> ( Pname ::= Pexp . )*
Pname -> 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(Term)        Communication (non-sync)
 | Comp               Computation (Prolog expression)
 | if(Comp, Pexp, Pexp) Conditional Expression
 | zero              Empty process (0 in CCS)
 | nil              Empty computation
PortMap -> [Port / Port (, Port / Port)*]
PortList -> { Port (, Port)* }
Term -> PrologTerm
Comp -> PrologPredicate
```

Port -> PrologAtom

Pname is a parameterized process name, represented as a Prolog term; *Comp* is a computation, e.g., X is Y+1. Process *in(Port,Term)* inputs a value over port *Port* and unifies it with term *Term*; *out(Port,Term)* outputs term *Term* over port *Port*; Process *action(Term)* specifies an action that is represented by *Term* and used for non-synchronous communication. Process *if (Comp, Pexp, Pexp)* behaves like the first *Pexp* if computation *Comp* succeeds and otherwise like the second *Pexp*. Operation 'o' is sequential composition; ' | ' is parallel composition; '\#' is nondeterministic choice; '@' is re-labeling where *PortMap* is a list of substitutions; and ' \backslash ' is a restriction where *PortList* is a list of port names. Recursion is provided by a set of process definitions, *Pdef*, of the *form Pname ::= Pexp*.

As an example, consider the specification of the Alternating Bit Protocol \cite{T96} 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 {
        NSeq is 1-Seq;
        sendnew(AckIn, DataOut, NSeq)
      }
      %% 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 {
      NSeq is 1-Seq;
      action(recv);
      out(AckOut, RecSeq);
      receiver(DataIn, AckOut, NSeq)
    }
    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).
```

The process *medium* represents a noisy channel. The *sender* process sends a packet to the channel and waits for an acknowledgement. Upon timeout, it resends the packet. The *receiver* process receives a packet from the channel and sends an acknowledgement back. The *abp* process is the parallel composition of the previously described processes.

Notice that using this notation, we can define the instantiation processes for the following types:

- Basic processes – the processes were already defined in the previous sub-section;
- Recursive processes – this processes have the form IPname ::= … IPname …, where "IP" denotes an instantiation process;
- Sequential processes – these processes use the prefix operator "o" to define the sequence of two instantiation processes IP1 and IP2: "IP1 o IP2";
- Choice instantiation process – the choice between instantiation processes IP1 and IP2 is denoted by "IP1 # IP2";
- Parallel Composition instantiation process – this processes use the parallel composition operator "|"; when there is a choice between executing an instantiation task P1 and doing nothing, this process is denoted by "P1 | zero"
- Combined instantiation processes use two or more of the processes described above.

Analysis. As we have previously mentioned, we will verify structural framework properties using the XSB Prolog deductive facilities and verify the process-based instantiations properties expressed in the μ-calculus temporal logic using the XMC model checker.

The μ-calculus temporal logic is a modal calculus whose semantics are 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 V F | F /\ F | <A> F | [A] F

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

Temporal properties, such as deadlock and drop package, can be described in μ-calculus and can be checked against the model of the Alternating Bit Protocol by XMC. The description of these two properties is as follows:

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

%% A packet can be lost without being received
drop_packet += <sendnew>lost V <->drop_packet.
lost += <sendnew>tt V <-recv>lost.

This temporal logic also allows us to give the semantics for the "**requires**" in terms of constraints on the order or exclusion of process instantiations. For example, if in a process IP1 there is a "**requires**" clause stating "**before** IP2", we need to introduce an instantiation constraint. Furthermore, if there is a "**requires**" clause stating "**exclusive**" IP2, we also need to introduce an instantiation constraint.

III. Case Study

In this section, we first describe the DrawingTool framework and then follow the phases of our process-based instantiation approach to represent, instantiate and analyze this system. The analysis is performed by checking properties about the framework instantiation process.

A. The DrawingTool Framework

To illustrate the approach, we will present a set of examples based on a framework called DrawingTool, a whitebox framework developed as part of the ARTS project , an on going project at PUC-Rio, in order to provide the drawing characteristics of the 2GOOD case tool much like HotDraw [18]. These drawing characteristics are illustrated in Figure 2. In summary, this framework has the following features:

- Figure Drawing – Provides architecture to draw, move and erase figures in a drawing. The new figures are introduced through subclassing. This is a mandatory feature in the DrawingTool design.
- Drawing Persistency – Provides persistency of the drawing. This is an optional feature in the DrawingTool design.
- Drawing Exportation – Supports the export of a drawing. This is an optional feature in the DrawingTool design.

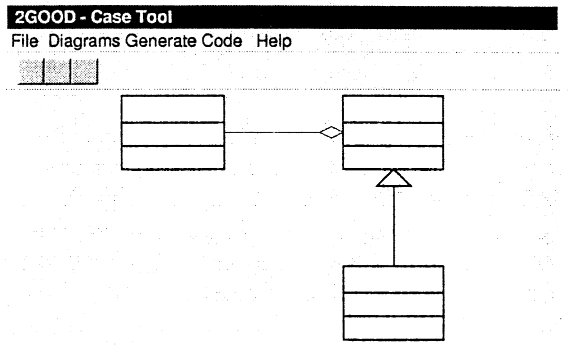
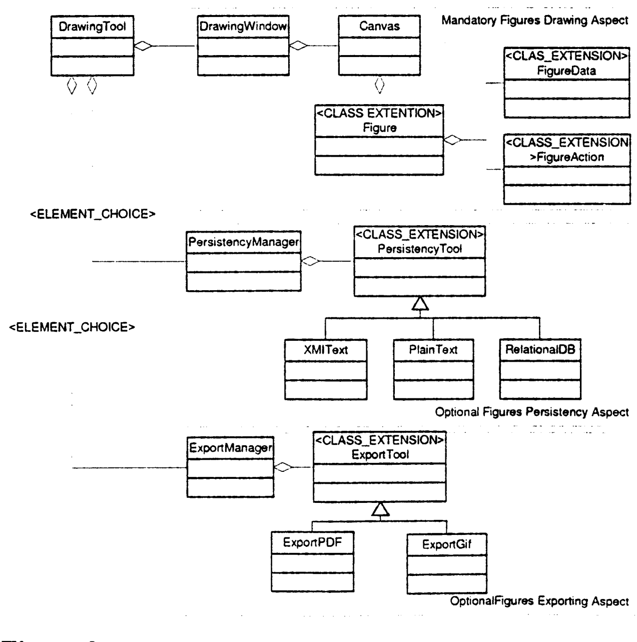


Figure 2

B. Representation

In order to use our approach we need to decorate the UML class diagrams that represent the whole structure of the DrawingTool framework, which is the reusable artifact. These decorations provide the representation of the framework extension points and are represented using UML extension mechanisms such as stereotype and tagged-value. The extension points are the subject of the instantiation process described in section II.



DrawingTool
DrawingWindow
Canvas
Mandatory Figures Drawing Aspect
<CLAS_EXTENSION> FigureData
<CLASS EXTENTION> Figure
<CLASS_EXTENSION >FigureAction
<ELEMENT_CHOICE>
PersistencyManager
<CLASS_EXTENSION> PersistencyTool
<ELEMENT_CHOICE>
XMIText
PlainText
RelationalDB
Optional Figures Persistency Aspect
ExportManager
<CLASS_EXTENSION> ExportTool
ExportPDF
ExportGif
OptionalFigures Exporting Aspect

Figure 3

In Figure 3 we show the decorated class diagram for the DrawingTool framework. Note that some classes are decorated with the stereotype CLASS_EXTENSION. This form of decoration allows us to specify the specific type of instantiation process associated with a class.

The framework extended UML diagram presented in Figure 3 is translated in a straightforward way to XSB Prolog using the design representation primitives in Section II-C.

As stated in [2] "the process of framework instantiation is achieved with the introduction of application specific increments (ASI for short) to the framework extension points". In the DrawingTool framework these ASIs are represented by the introduction new figures types and the choice of persistence and export aspects.

In order to achieve these ASIs, during the instantiation process the re-user will be required to provide the application specific elements. For example, in the DrawingTool framework the re-user needs to provide the kind of figures the application will have, so that they can be subclasses of the Figure Class.

C. *DrawingTool Instantiation Process*

In this Section we present the process descriptions that are associated with each of the three DrawingTool extension points. The resulting process descriptions indicate how DrawingTool can be instantiated and what tasks should be performed to accomplish a specific instantiation. There are choices related to which processes can be used in the instantiation process.

Note that in Figure 3 there is one mandatory aspect (Figures Drawing) and two optional aspects (Figures Persistency and Figures Exporting). The DrawingTool instantiation 'recipe' is called the DrawingTool Cookbook and is defined in what follows as an instantiation process using the language constructs that we have defined in Section II to represent instantiation tasks:

```
cookbook DrawingTool
// Recipe Instantiate Figures Aspect
// Choose the Persistency Aspect
element_choice (DrawingWindow.thePersistencyMan);
// Choose the exporting aspect
element_choice (DrawingWindow.theExportMan);
// Recipe for adapting the drawing features, such as
specifying the figure types.
        // Figure Creation Loop
        loop
        // Creates a new figure class in the design space
        figClass = class_extension (Figure);
        // Creates the Data representation for the created
        figure
        figDataClass = class_extension (FigureData);
        // Creates the actions representation for the created
        figure
        figAction = class_extension (FigureAction);
        // Adapts the figure data creation using the Factory
        Pattern. The sequence corresponds to
        (TheConcreteCreatorClass, TheFactoryMethod,
        TheConcreteProduct)
        pattern_method_extension (Figure, figClass,
        createData , Factory, (figClass, createData,
        figDataClass);
        // Adapts the figure actions creation using the
        Factory Pattern. The sequence corresponds to
        (TheConcreteCreatorClass, TheFactoryMethod,
        TheConcreteProduct)
        pattern_method_extension (Figure, figClass,
        createAction , Factory, (figClass, createAction,
        figActionClass);
        // Adapts the saving structure of the new figure.
        Requires the persistency aspect.
        Method_extension (Figure, figClass,save)
        requires (PersistencyManager);
        // End Figure Creation Loop
        end_loop;
// End Figure Aspect

// Recipe Instantiate Persistency Aspect
        // Adapts the persistency algorithm. It depends on
        the type of database used. If the DB type is note
        provided, creates a new one by subclassing
        PersistencyTool.
        select_class_extension (PersistencyTool) #
        class_extension (PersistencyTool) ;
        // Adapts the automatic persistency rate.
        value_selection(DrawingTool,
        autoPersistencyRate,(0,5,10,30))
        requires (PersistencyManager);
// End Persistency Aspect

// Recipe Instantiate Export Aspect
// Performs a loop in order to adapt several methods of
exportation
LOOP
        // Adapts the exportation algorithm.
        Select_class_extension(ExportTool)#
        class_extension (ExportTool) ;
// END_Loop;
// End Export Aspect
```

D. *Instantiation Analysis*

Instantiation errors can be difficult to detect by visual inspection. The goal of the framework instantiation analysis is to assure that the framework evolution is valid. In this way, we are able to changes the architecture based on explicit instantiation processes and activities without compromising the system integrity, invariants, and general constraints.

In order to analyze the framework we first need to translate the DrawingTool Framework Cookbook presented in the previous Section to XL. This translation is straightforward. For example, the instantiation process related to Recipe Instantiate Export Aspect is given by:

RecipeInstantiateExport(ExportTool) ::=
((select_class_extension(ExportTool) # class_extension(ExportTool)) # zero) o RecipeInstantiateExport(ExportTool)

Note that the loop was translated into a recursive XL process. Also, note that there is a choice between the processes *select_class_extension* and *class_extension.*

After translating the instantiation processes to XL, the resulting formal specifications allow us to reason about the framework instantiations. We can also use the model checking techniques supported by XMC to check, for example, if a certain instantiation task obeys its related constraints. In this way, we are able to find if the instantiations we are performing are valid. Our analysis is based both on the framework extensions and the processes used to integrate these extensions. In the following we illustrate the kinds of analysis our approach allows developers to perform about the instantiation process.

Basic type checking and structural properties. In the following we illustrate the kinds of properties that can be verified using our approach. We first describe some of the properties that can be checked using XSB Prolog. The analysis of basic properties includes type checking in order to verify whether extended elements have interfaces that match the associated elements, which they are extending. As an example of structural properties that can be checked, we can verify whether the classes that were created in the instantiation process are all correctly connected to the framework core.

Safety and Liveness Properties. We also analyze the framework instantiation processes using XMC. Behavioral properties about these properties include safety (always) and liveness (eventually) properties. A simple liveness property *<IP> zero* can be used to verify that an instantiation process *IP* eventually reaches the zero process or stops. The following property about the process RecipeInstantiateExport(ExportTool) can be easily verified: *<RecipeInstantiateExport(ExportTool)> zero.*

Another form of liveness property can check if the instantiation process contains a certain sequence of tasks. For example, in order to show that if the process *select_class_extension(ExportTool)* happens eventually, then the process *RecipeInstantiateExport(ExportTool)* will also happen eventually. In order to assess this property we prove that the following property holds :

< select_class_extension(ExportTool)>
<RecipeInstantiateExport(ExportTool)> tt

More complex sequences of processes related to different 'recipes' can also be represented and checked using similar expressions.

Invariants. Invariants that must hold during the instantiation process independent of the tasks that are performed can be represented by the following property:

[_] invariant_expression,

where, if the property is based on the states reached during the process, we may write <_> *variable = constant.*

Deadlocks. We can also confirm the fact that the instantiation process is free of deadlocks by proving the following property:
deadlock += [-] ff ∨ <-> deadlock.

Process Pre- and Post-Conditions. Pre-and post-conditions for an instantiation process can also be represented using our formalism.

Other Properties. Our approach can also be used to perform many other kinds of analyses on the *DrawingTool* framework extension. Examples include extension point constraint checking "can an instantiation related to an extension point be performed only if a pre-condition holds?", extension process execution "does a process state satisfy a certain constraint, a post-condition, after the process has executed?", instantiation process order anomalies "does a process IP1 always happens after the instantiation process IP2 is performed?" reachability "is process IP2 eventually executed if an instantiation process IP1 is executed?", and mutual exclusion "an instantiation process IP1 is never executed at the same time that an execution process IP2 is executed.".

The analysis is limited by the kinds of properties that can be expressed using μ-calculus and the kinds of processes that can be represented by XL, a variant of the Milner's CCS process algebra. However, both formalisms are highly expressive. As a temporal logic, the μ-calculus is more expressive than linear temporal logic (LTL) and branching time logics such as CTL and CTL*.

IV. Related Work

Framework and Pattern Documentation and Analysis. Some effort to achieve framework documentation and systematic instantiation has been described in [18,19,5,20]. The Cookbook [6] and Hook [19] approaches describe the instantiation process using a natural-language form of description, structured in a special way. The problem with these approaches is the lack of a formal representation for both the design and reuse scripts. There is also no way to verify the correctness of the final design according the original framework premises.

[6] proposes an extension to the UML notation in order to provide more accurate framework design. Although its extension points are well represented and it uses a Prolog knowledge base to reflect its design adaptations, it is still difficult to obtain automatic instantiation because of the lack of an understandable output.

Various design pattern recovery methods and tools [21,22,23] are related to our work in the sense that all recovered patterns can be modeled and checked for anomalies in their compositions. The correction of these anomalies could be used to complete the reengineering tasks.

Framework and Pattern Specification and Verification. Formalizing design patterns and architecture patterns has been proposed in [24,25]. Although Mikkonen [25] has discussed the composition of two design patterns based on a formal method, his approach relies on a specific specification language (DisCo). Correctness depends on the refinement correctness of this language since the composition is achieved in terms of refinement. Our approach emphasizes specifying design components and their compositions, and checking the properties by a model checker. Moreover, Mikkonen's approach focuses on formalizing design patterns, whereas our work deals with a more general approach based on design components [11,30] and their composition.

Pal [26] investigated rule and constraint approaches for realizing design patterns. However, in his work the property checking is performed at implementation level. He did not discuss the interactions among different design patterns when they were integrated together.

Other work on tool support for object-oriented patterns [27] also discussed constraints on patterns. Nevertheless, they worked on single pattern constraints at implementation level. Our work emphasizes the interactions among different patterns when they are integrated.

Automated Verification through Model Checking. There is an increasing interest in modeling and analyzing software by various formalisms, and checking properties or finding errors through model checking [8,12,28]. Numerous examples can be found in various domains such as requirement analysis [29], distributed cache coherence analysis [31], word processor design analysis [32], mobile IP protocol analysis [33], CAD algorithm analysis [34], real-time operating system kernel analysis [35], and Java meta-locking algorithm analysis [36].

V. Conclusions and Future Work

In this paper we have presented an approach for process-based representation and analysis of framework instantiation. The approach is based on explicit instantiation and reuse processes and automated verification techniques. We illustrate these representation and analysis techniques through a case study related to the instantiation of the DrawingTool framework.

Discovering composition errors during the instantiation process using architectural and design level abstractions is certainly much easier than relying on code, because a small piece of design may be mapped to thousands of lines of implementation code. Instantiation errors may be hidden in complex implementation structures and are very costly to analyze or detect. Furthermore, at the implementation stage design errors are very costly to modify.

Our approach has several advantages. First, it allows us to find errors in the instantiation process using a level of abstraction where errors are cheaper to correct. Second, it provides mechanisms to achieve automated verification of framework instantiation properties. Third, it promotes reuse, since the generic representations of framework design elements can be stored in a repository and retrieved for instantiation and integration in a specific application. Fourth, as the instantiation processes can be aggregated forming complex and significant additions to the initial framework, the analysis can scale up incrementally to deal with large frameworks. For this reason, the state-explosion problem in model checking can be addressed by incrementally modeling the instantiation processes and checking the process combinations. Fifth, changes of the framework design elements and the instantiation processes can be achieved by simply modifying some arguments of their parameters.

Our analysis approach is limited to the kinds of properties that can be proved using Prolog and the highly expressive μ-calculus temporal logic. Based on the results of the experiments we have done so far, these underlying deductive facilities seem to be adequate. Besides verifying structural and process-based instantiations, we are currently defining other classes of properties that we can use in our analysis of framework instantiations. These classes may include properties about (real-)time, event ordering and access control.

A more comprehensive infrastructure needs to be defined for our approach, including process management dealing with process state such as context and preferences, pro-active process execution and guidance where the system may offer suggestions to help the user accomplish goals, process coordination, facilities to deal with pending processes and corrective hints, and support for instantiation process definition such as some form of system visualization or snapshot and finally, processes that have well defined interfaces and metadata specification such as the support of automatic process composition and configuration and where legacy elements can be wrapped to support these interfaces.

We are also working on techniques to map the XL counter-examples back onto the original structural and process descriptions. We are assessing 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 instantiation process descriptions need to be revised under the light of the counter-examples.

This paper also indicates that it could be highly beneficial to pursue further research on how the structural framework design changes may affect the behavior of the framework, and how these changes can affect various properties of the instantiation process. These analyses can provide guidance about important framework instantiation activities as well as be documented to provide information on the rationale behind these activities.

VI. Acknowledgments

This work is being supported in part by the NSERC Strategic project grant funded by the National Sciences and Research Council of Canada and the National Research Council of Brazil (CNPq).

VII. References

[1] Pree, W., Design Patterns for Object-Oriented Software Development, Addison-Wesley Publishing Company, 1995.
[2] Fayad, M.E., Schmidt, D.C., Johnson, R.E., Domain-Specific Application Frameworks, Wiley Computer Publishing, 1999.
[3] Frakes, W.B., Biggerstaff, T.J., Prieto Dias, R., Matsumura, K., Schafer, W., Software Reuse: is it delivering? ICSE, pp. 52-59, 1991.
[4] Booch, G., Rumbaugh, J., Jacobson, I., The Unified Modeling Language User Guide, Addison-Wesley, 1999.
[5] Marcus Felipe Montenegro Carvalho Da Fontoura. A Systematic Approach to Framework Development. Ph.D. Thesis, Computer Science Department, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), 1999.
[6] Krasner, G.E., Pope, S.T., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, Journal of Object-Oriented Programming 1(3), 1998.
[7] XSB, The XSB Logic Programming System, Version 2.1. Available from http://www.cs.sunysb.edu/ \sim sbprolog, 1999.
[8] Ramakrishna, Y.S., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Swift, T., Warren, D.S., Efficient Model Checking Using Tabled Resolution, Proceedings of the 9th International Conference on Computer Aided Verification (CAV), Haifa Israel, LNCS1243, Springer-Verlag, July 1997.
[9] McMillan, K. L., The SMV system DRAFT, Carnegie-Mellon University, pp. 1-24, February 1992.
[10] Clarke, E.M., Emerson, E.A., Sistla, A.P., Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, ACM Transactions on Programming Languages and Systems,8(2), pp. 244-263, April 1986.
[11] Dong, J., Model Checking the Composition of Hypermedia Design Components, Proceedings of the 10th CASCON, Toronto Canada, pp. , November 2000.
[12] Ramakrishnan, C.R., Ramakrishnan, I.V.,Smolka, S.A., XMC: A Logic-Programming-Based Verification Toolset, Proceedings of the International Conference on Computer Aided Verification (CAV), LNCS1855, Springer-Verlag, July 2000.
[13] D. Kozen, Results on the Propositional mu-calculus, Theoretical Computer Science 27, pp. 333-354, 1983.
[14] Milner, R., Comunication and Concurrency, International Series in Computer Science. Prentice Hall, 1989.
[15] Alencar, P.S.C., Cowan, D.D., Dong, J., Lucena, C.J.P., A Pattern-Based Approach to Structural Design Composition, Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference (COMPSAC), Phoenix USA, pp. 160-165, October 1999.
[16] Dong, J., A Transformational Process-Based Approach to Object-Oriented Design, Master's Thesis, Computer Science Department, University of Waterloo, 1997.
[17] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1995.
[18] Johnson, R., Documenting Frameworks Using Patterns, Proceedings of OOPSLA'92, ACM/SIGPLAN, New York, 1992.
[19] Froehlich, G., Hoover, H.J., Liu L. and Sorenson, P.G. Hooking into Object-Oriented Application Frameworks, Proc. 19th Int'l Conf. on Software Engineering, Boston, May 1997, 491-501.
[20] Ortigosa, A., Campo, M., Smartbooks: A Step Beyond Active-Cookbooks to Aid in Framework Instantiation, Technology of Object-Oriented Languages and Systems 25, IEEE Press, June 1999.
[21] Rudolf K. Keller, Reinhard Schauer,S\'{e}bastien Robitalille, Patrick Page, Pattern-Based Reverse-Engineering of Design Components, Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA, pp. 226-235, May 1999.
[22] Daniel Jackson, Allison Waingold, Lightweight Extraction of Object Models from Bytecode, Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA, pp. 194-202, May 1999.
[23] Christian Kramer, 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, November 1996.

[24] Alencar, P.S.C., Cowan, D.D., Lucena, C.J.P., A Formal Approach to Architectural Design Patterns, Proceedings of the Third International Symposium of Formal Methods Europe, pp. 576-594, 1996.

[25] Tommi Mikkonen, Formalizing Design Pattern, Proceedings of the 20th International Conference on Software Engineering, pp. 115-124, 1998.

[26] Pal, P., Law-Governed Support for Realizing Design Patterns, Technology of Object-Oriented Languages and Systems (TOOLS), USA, pp. 25-34, July 1995.

[27] Florijn, G., Meijers, M., Winsen, P., Tool Support for Object-Oriented Patterns, Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), pp. 472-495, June 1997.

[28] E. M. Clarke, J. M. Wing, Formal Methods: State of the Art and Future Directions, ACM Computer Surveys 28(4), December 1996.

[29] Joanne M. Atlee, John Gannon, State-Based Model Checking of Event-Driven System Requirements, IEEE Transactions on Software Engineering, 19(1), pp. 24-40, January 1993.

[30] Rudolf K. Keller, Reinhard Schauer, Design Components: Towards Software Composition at the Design Level, Proceedings of the 20th International Conference on Software Engineering, pp. 302-311, 1998.

[31] Jeannette M. Wing, Mandana Vaziri-Farahani, A Case Study in Model Checking Software Systems, Science of Computer Programming, 28, pp. 273--299, 1996.

[32] Daniel Jackson, Craig A. Damon, Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector, IEEE Transactions on Software Engineering, 22(7), pp. 484--495, July 1996.

[33] Zhe Dang and Richard A. Kemmerer, Using the ASTRAL Model Checker to Analyze Mobile IP}, Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA, pp. 132-141, May 1999.

[34] David S. Keyes and Laura K. Dillon and Moon Jung Chung, Analysis of a Scheduler for a CAD Framework, Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA, pp. 152-161, May 1999.

[35] John Penix, Willem Visser, Eric Engstom, Aaron Larson, Nicholas Weininger, Verification of Time Partitioning in the DEOS Scheduler Kernel, Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, pp. 488-497, June 2000.

[36] Samik Basu, Scott A. Smolka, 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), pp. 342-350, April 2000.