Assisting Framework Instantiation:
Enhancements to Process-Language-based Approaches

Technical Report CS-2005-25 (August 2005)
School of Computer Science, University of Waterloo

Marcilio Mendonca
University of Waterloo, Canada
marcilio@csg.uwaterloo.ca

Paulo Alencar
University of Waterloo, Canada
palencar@csg.uwaterloo.ca

Toacy Oliveira
PUC-RS, Brazil
toacy@inf.puc-rs.br

Donald Cowan
University of Waterloo, Canada
dcowan@csg.uwaterloo.ca

Abstract

Application frameworks have been successfully used as valuable tools to improve software quality while reducing development efforts. Nevertheless, frameworks still face important challenges in order to be widely adopted. In particular, framework instantiation is still a painful task requiring application developers to understand the intricate details surrounding the framework design. Some approaches to alleviate this problem have already been proposed in the literature but they are usually either just a textual cross-referenced document of the instantiation activities or too tied to technology or specific application domains. In this paper, we present the results of our latest investigations to improving our approach to framework instantiation. In particular, we discuss a process language we have developed to guide framework instantiation explicitly, and the most recent updates we have made to improve the language expressiveness. Furthermore, we present a case study used to evaluate our approach and to identify current and future extensions.

Keywords: frameworks, instantiation, software process, software design, transformation.

1 Introduction

Framework concepts have been successfully employed as important tools to achieve software reuse [6]. At the same time they reduce developments efforts and increase the overall quality of produced software systems. ET++ [24], MacApp [25] Hotdraw [30], MFC [26], just to name a few, are important examples of early frameworks that helped demonstrate the feasibility of a framework-centered development approach. They were able to capture common features successfully and represent the variability of a family of applications within a specific domain. There are now a large number of frameworks that have been developed for a variety of different purposes including CORBA [23] (middleware for distributed systems), JADE [20] (agent systems), Struts [12] (web applications), JBoss-AS [27] (enterprise applications), and JUnit [22] (application testing).

However, as frameworks become popular their weaknesses as well as their strengths are becoming apparent. In particular, framework instantiation is still a painful task because application developers must understand the intricate details surrounding the framework design. Thus, instantiation of a specific application can often be a slow and costly process. For some frameworks such as the MFC it may take up to 12 months for an application developer to be highly productive [6]. Thus, the instantiation process is a time-consuming activity which is counter to one of the most valuable prop-

erties of reuse, i.e., significant shortening in development time.

Some approaches to alleviate the framework instantiation problems we mentioned have been proposed in the literature [5],[7],[8],[9],[29]. However, they are normally either just a textual cross-referenced documentation of the instantiation activities or too closely to technology or specific application domains.

In this paper we initially give an overview of our approach to framework instantiation. We present RDL (Reuse Definition Language) a process language we have created to represent framework instantiation activities explicitly. RDL along with xFIT, our supporting instantiation tool, operates on UML [17] models through transformations in order to produce valid application instances. Following this description, we present the latest enhancements we have made to our RDL in order to improve its expressiveness. Finally, we discuss a case study we conducted to assess how our framework improved framework instantiation.

The sections of this paper are organized as follows. Section 2 presents our approach to framework instantiation and its latest enhancements. Section 3 depicts the case study we carried out in order to assess our approach properly. In section 4 we discuss how our approach improved framework instantiation in the light of our case study evidences. Section 5 includes some related work, and finally Section 6 presents our conclusions and future work.

2 Our Approach to Framework Instantiation

A typical framework adaptation has two phases: i) understanding the overall rationale behind the framework design; ii) extending the framework flexible points according to specific requirements in order to produce application specific increments (ASI) [4].

As we have mentioned, the first phase has been supported by some framework documentation approaches. Basically, they describe the purpose of the framework, its major design elements, their relationships, how the flexible points can be adapted to produce applications, and provide some examples. For example, in the cookbook approach [8], recipes are used to explain how a certain extension point can be adapted. Recipes can reference each other, thus helping application developers to understand better how the hotspots (and the design elements they represent) are interrelated.

Our approach complements framework documentation techniques, in particular cookbooks. It closes the gap left by purely text-based approaches by providing means to represent instantiation activities explicitly. Our approach consists of a process language, RDL, that allow framework developers to represent adaptation steps, and a supporting tool, xFIT, that operates on UML models by transforming a framework's class diagrams into application class diagrams based on application developer's inputs. Figure 1 below depicts our approach.

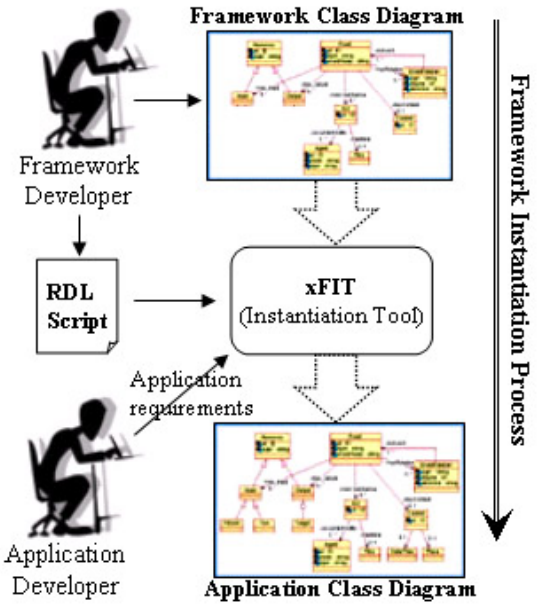

Figure 1: Overview of Our Approach

The steps required to instantiate a framework using our approach consists of:

- The framework developer provides a framework class diagram conforming with the XMI format (an XML file representing the model);
- The framework developer provides an RDL script containing the framework instantiations steps;
- The application developer runs xFIT providing it with the RDL script and the framework UML class diagram. Likewise, the application developer provides feedback according to specific application requirements.

- At the end of the generation process xFIT will run validation tasks and report the errors encountered (if some exist). Otherwise, a UML class diagram is produced including the framework and the specific application instance classes.
- The application developer can then use a Case tool to open the application model and generate stubs for the classes produced. By filling out the stubs with appropriate code the process is ended.

2.1 The Process Language (RDL)

RDL is a process language that aims at providing mechanisms for framework developers to represent instantiation tasks explicitly. RDL is programming-language and framework-domain independent and manipulates design elements expressed in UML. RDL abstractions have been proposed based on the cookbook approach and exploit the use of design patterns [1].

In the next sections we describe the main construct of the RDL and the latest enhancements we have made to increase the language expressiveness. Since it is not the purpose of this paper to be an RDL reference manual we suggest reading [2] for a more detailed description of the language.

RDL Main Structure- RDL higher level constructs are represented by cookbooks, recipes and patterns. AN RDL cookbook contains a set of RDL recipes. RDL recipes embody instantiation tasks related to particular variable aspect of a framework architecture. RDL Patterns describe recurring instantiation steps encountered during a framework adaptation (e.g. design patterns).

RDL can be used to produce two types of artefacts: RDL scripts and Pattern Libraries. A general structure of an RDL script is shown in Table 1.

```
COOKBOOK myCookBook
  RECIPE main
    …
    CALL_RECIPE( R1, (…) );
    …
  END_RECIPE
  RECIPE R1(…)
    …
  END_RECIPE
  …
END_COOKBOOK
```

Table 1: General structure-RDL Script

In an RDL script, at least one recipe must be named *main* representing the cookbook start point. Recipes can call each other, receive parameters and return values in a way similar to functions in procedural languages.

RDL Pattern Libraries describe instantiation patterns, i.e., recurring instantiation tasks. Since some design patterns exhibit an abstract and a concrete part (e.g. Template Method, Abstract Factory, and Strategy) they can be properly used to expose framework hotspots. Therefore, design patter instances can be represented as RDL Patterns. Pattern Libraries represent an enhancement we have made to our approach. The general structure of an RDL Pattern Library can be found in Table 2.

```
PATTERN_LIBRARY  myPatternLibraryName
  PATTERN Pattern1(…)
    …
  END_PATTERN
  PATTERN Pattern2(…)
    …
  END_PATTERN
  …
END_PATTERN_LIBRARY
```

Table2: General structure-RDL Pattern Library

Pattern libraries are normally stored in files with the *.rdp* extension (reuse definition pattern). UML class models are expected to conform to the XMI format and are stored in *.xmi* files. In the current version of RDL only one RDL script *(.rdl)* is allowed to specify the instantiation steps of a framework. There is still no way to import and combine RDL scripts. This has been left as a possible enhancement for future versions of our language.

RDL Types- In order to keep the syntax of the language simple, the previous versions of RDL did not considered data types explicitly. However, as we used the language in practical situations the need for a strong typed-language became apparent. Thus, the types now encountered in RDL are basically those found in UML class diagrams plus some additional ones to represent strings, numbers, booleans and list of types (Table 3). Each type has a set of associated operations and attributes that allow framework developers (RDL script users) to make proper references to model elements.

RDL Type	Description
STRING	Represent Strings in RDL

NUMBER	Represent Numbers in RDL
BOOLEAN	Represent Booleans in RDL
PACKAGE	Represent UML Packages
CLASS	Represent UML Classes
METHOD	Represent UML Class Methods
ATTRIBUTE	Represent UML Class Attributes
Lists	Represent lists (vectors) in RDL

Table 3: RDL Types

RDL commands fall into three categories: Basic, Instantiation, and Pattern Commands. Following we discuss each one of the categories.

RDL Basic Commands- The basic commands provide low-level facilities to manipulate the framework design elements. For instance, new classes, methods or attributes can be created and added to UML class diagram models. Table 4 below illustrates some of the RDL basic commands.

Description	**Basic Command**
Class creation	NEW_CLASS(…)
Method creation	NEW_METHOD(…)
Attribute creation	NEW_ATTRIBUTE(…)
Inheritance	NEW_INHERITANCE(…)
Selection	IF (e) … [ELSE ...] END_IF
Repetition	LOOP (e) … END_LOOP
Assignment	Var = expression

Table 4: Main RDL Basic Commands

Instantiation Commands- RDL Instantiation Commands increase the level of abstraction by combining basic commands into single tasks. Basically, RDL Instantiation Commands represent object-oriented reuse activities such as extending a class, overriding a method, and assigning a value to a class attribute. Table 5 depicts the main Instantiation Commands.

Description	**Instantiation Command**
Class Extension	CLASS_EXTENSION(…)
Method Extension	METHOD_EXTENSION(…)
Value Assignment	VALUE_ASSIGNMENT(…)
Value Selection	VALUE_SELECTION(…)

Table 5: Main RDL Instantiation Commands

Pattern Commands- The highest level statements in RDL are represented by Pattern Commands. Pattern Commands allow framework developers to reuse a set of recurring instantiation activities previously specified. In previous versions of RDL, Patterns Commands were represented by the *Pattern Class Extension* and *Pattern Method Extension* commands. These commands required specific types to be passed as input parameters in order to be used properly. We decided to simplify the language support for patterns by defining a single command for an RDL Pattern call. No parameters are required and it is up to the framework developer to define how patterns will be properly described. Table 6 describes the RDL command to call an RDL Pattern.

Description	**Pattern Command**
Pattern call command	CALL_PATTERN(..)

Table 6: RDL Pattern Call Command

In the following we illustrate the implementation of an RDL Pattern Library including an implementation for the *Factory Method* design pattern, and an RDL Script benefiting from the RDL Pattern implementation.

```
PATTERN_LIBRARY GammaPatterns

  PATTERN FactoryMethod(
      IN  absCreatorName  : STRING ,
      IN  facMethodName  : STRING ,
     INOUT concreteCreatorClass : CLASS  )

      -- Create Concrete Creator
      IF (concreteCreatorClass = NIL)
        concreteCreatorClass =
           CLASS_EXTENSION(
                absCreatorName, ? );
      END_IF
      -- Extends Creator Factory Method
      METHOD_EXTENSION( absCreatorName,
      facMethodName, concreteCreatorClass );
  END_PATTERN
END_PATTERN_LIBRARY
```

```
COOKBOOK myCookBook
  RECIPE main
    conCreator : CLASS;
    CALL_PATTERN(
       FactoryMethod, (
         "AbstractView",
         "createAlarm", conCreator));
END_RECIPE
END_COOKBOOK
```

2.2 The Framework Instantiation Tool (xFIT)

Our approach is supported by an instantiation tool known as xFIT (Framework Instantiation Tool). xFIT provides a runtime environment for RDL

scripts. The framework class diagram and an RDL script are taken as inputs and based on application developers feedback xFIT generates the application instance class diagram. xFIT performs validation tasks over the design elements produced to ensure that its structure is regular and well-formed. As an example, xFIT certifies that all abstract classes and methods have been resolved in the final design since all the hotspots must have been handled.

3 The Case Study

Our case study aimed at assessing our approach to framework instantiation. In particular, we were interested in comprehending how an explicit approach would improve framework instantiation when contrasted with purely text-based approaches (e.g. [5],[7],[8],[9]). Yet, it was not our intention to address any specific text-based approach but rather to treat them as a category. In summary, our goal was to answer the following questions:

- How did our approach improved framework instantiation when compared with purely text-based approaches?
- Which issues in framework instantiation were not addressed by our approach? How can we support them in the future?

In the next sections we provide a detailed description of our case study. We begin by presenting the framework we have chosen as the target of our study, then we discuss how previous instantiations were carried out, and finally we give details on how the case study was conducted.

3.1 The Framework (REMF)

The Real-time Event Monitoring Framework (REMF) (see Figure 2) has been developed and maintained by our group to be applied to the monitoring of real-time events. Originally, the intention was to develop an application to monitor real-time events produced by one of our server applications. However, we realized that some other applications and projects within our research program could also take advantage of our monitoring tool so we decided to develop it as a framework.

Several factors influenced our choice in favor of the REMF as our case study:

i) the quality of its architecture strongly supported by design patterns and the MVC [28] pattern;
ii) the maturity of its architecture after several successful adaptations;
iii) our previous experiences adapting its design to address specific application requirements, which provides insight in assessing the improvements brought by our approach;
iv) and finally, its straightforward architecture that allow us to concentrate our efforts on discussing our approach rather than on the specific design solutions adopted.

The domain covered by the REMF is quite wide but we identified some sub-domains that can benefit from its design:

- Business-rules monitoring tools (ex.: tools to monitor the business-rules of server applications in general, such as those for credit cards, health services, debit cards, loyalty cards, etc.; also, web-servers, service-oriented applications, etc.)
- Application monitoring tools (ex.: tools to monitor the performance of server applications such as the number of resources allocated, time spent in processes, exceptions, etc.)
- System monitoring tools (ex.: tools to monitor system events such as memory and CPU performance, SNMP events, etc.)

REMF Flexible Points- REMF was written in Java 1.3 [21] using Eclipse 3.0.1 [16] as the Java compiler/debugging tool and the Eclipse plug-in Omondo EclipseUML 2.0 [13] as the UML Case tool. The framework contained 64 classes distributed across 7 packages. Eighteen (18) of the classes were abstract and 46 were concrete revealing the framework developer's commitment to providing a rich set of ready-to-use functions (frozen spots) shared among all application instances. REMF architecture was supported by 7 design patterns: Factory Method, Strategy, Observer, Template Method, Abstract Factory, Producer/Consumer, and Iterator. The MVC pattern constitutes the core of the framework architecture exposing a variety of flexible points. Seventeen (17) hotspots were exposed by the REMF architecture. Seven (7) of them were mandatory and 10 were optional. It means that, at a minimum, an instance could be obtained by resolving the 7 mandatory hotspots and accepting the default be-

havior provided by REMF architecture for the other 10 optional ones. All hotspots have been implemented by using design patterns which allowed experienced programmers to understand rapidly the rationale behind them. Table 7 gives an overview of the REMF hotspots, its recipes, and the design patterns used.

Roughly, there were three major elements that can be customized in the REMF architecture: the *Event's Producer*, the *Filtering Capabilities*, and the *Alarms*.

Figure 2: REMF Instance Screenshot

The *Event's Producer* was in charge of producing the real-time events that would be displayed in the graphical user interface. For example, a given REMF instance may have decided to use a socket to connect to a server application in order to listen for events. *Event Producers* were implemented based on the Producer/Consumer design pattern but had their hotpots exposed by the Template Method design pattern. The Filtering Capabilities allowed instances to customize filtering options based on their requirements. *Filtering options* were based on the Template Method, Observer and Factory Method design patterns. Finally, *Alarms* could be raised whenever a certain filter expression was matched. Each REMF instance could customize the alarm action (algorithm) to be taken. For example, a particular implementation may have decided to play a short audio file as an alarm action. Alarm actions were based on the Strategy design pattern.

The REMF also provided a mechanism to *glue* the application specific increments with the framework. The *glue* mechanism used reflection as a means to achieve runtime composition (Java reflection) and was based on the Abstract Factory design pattern.

3.2 Running the Case-Study

In order to assess the effectiveness of our approach properly we decided to reason about and document the previous REMF instantiation experiences first. Then, we defined a set of specific requirements to motivate our case study and the need for a new REMF adaptation. Finally, we applied our approach to produce the application instance planned.

ID	Hotspot	Recipe	Type	Default Behavior	Design Pattern
HNA1	Creating an Alarm Object	Notification Alarms	Optional	No alarms are defined	Factory Method
HNA2	Specifying the Alarm Action	Notification Alarms	Optional	No alarms algorithm is defined	Strategy
HAF1	Handling Visual Filter Selection Changes	Application Filters	Mandatory	-	Template Method
HAF2	Creating a Visual Filter Panel	Application Filters	Mandatory	-	Factory Method
HAF3	Creating a Textual Filter Expression	Application Filters	Mandatory	-	Template Method
HTF1	Defining the Application Top-level Filter	Top-level Filter	Optional	No Event is Filtered	Template Method
HFS1	Specifying the Event Message's Fields to be Saved	Filters to Save	Optional	All Event's Fields are saved	Template Method

HEP1	Specifying the Events Message's Fields Meta Definitions	Events Producer	Mandatory	-	Template Method
HEP2	Preparing Producer Resources	Events Producer	Optional	No Action	Template Method
HEP3	Releasing Producer Resources	Events Producer	Optional	No Action	Template Method
HEP4	Producing New Events	Events Producer	Mandatory	-	Template Method
HEP5	Defining a New Message Producer	Events Producer	Mandatory	-	Factory Method
HEP6	Specifying the Appropriate Action when Producer Starts Producing Messages	Events Producer	Optional	No Action	Observer
HEP7	Specifying the Appropriate Action when Producer Stops Producing Messages	Events Producer	Optional	No Action	Observer
HEP8	Specifying the Appropriate Action when Producer is Repairing an Error During a Message Production	Events Producer	Optional	No Action	Observer
HEP9	Specifying the Appropriate Action when Producer Produces a New Message	Events Producer	Optional	No Action	Observer
HGL1	Instantiating a Factory to create MVC Components	Glue	Mandatory	-	Abstract Factory

Table 7: REMF Hotspots

Previous Adaptations

Prior to our case study the REMF had already been instantiated on other five occasions producing distinct application instances (see Table 8).

REMF Instance	**Hotspots Adapted**
Credit card monitoring tool *1 instance*	17
Loyalty card monitoring tool *1 instance*	14
Prepaid cell phone account recharge monitoring tool *3 instances w/ specific requirements*	13, 13, 15

Table 8: REMF Previous Instances

All previous adaptations were carried out based on the REMF cookbook which consisted of a document describing the framework hotspots and their proper adaptation. Since no active guidance or tool support was provided application developers were required to perform the appropriate instantiation steps *manually*.

Every REMF adaptation followed a top-down development approach. First, design models were manipulated, i.e., the framework UML class diagrams were extended, then class stubs were automatically generated based on the UML models, and finally, specific Java application code was added to the stubs using a Java editor.

Although the adaptation process just described seemed to be appropriately high-level and straightforward, it was indeed complex and problematic. In particular, the lack of automation to assist in finding and extending REMF hotspots in UML class diagrams made the process of producing design models for application instances intricate and lengthy. Application developers were required to navigate though the framework class diagram, identify the hotspots, differentiate mandatory and optional hotspots, and properly extend the framework classes and methods, without any active assistance. This scenario turned out to be even worse when time-to-market forces started to pressure the application's deployment.

In order to alleviate the lack of guidance and adequately cope with time-to-market issues some important decisions had to be made. First, the recipes in the REMF cookbook were written to reflect the framework main architectural components. In particular, the MVC components were used as a guide to specify the recipes in a way that resembled what was done in [28]. The primary goal was to minimize the need to manipulate design elements spread throughout the model, and consequently diminish the complexity associated with it, and thus have instantiation steps referring

to design elements located in a specific architectural module. We say that the recipes followed an *architecture-oriented* arrangement. Figure 3 depicts the recipes and their relationships for the first five REMF adaptations.

Second, we decided to postpone detailed hotspot resolution such as method extensions to a later phase and let the design adaptation steps focus more on simple tasks such as class extensions.

This strategy appeared to be effective since at source code manipulation time the compiler is a helpful tool in identifying mandatory hotspots. For instance, a compiler will raise an error whenever a super-class abstract method is not overridden by a sub-class (mandatory hotspots).

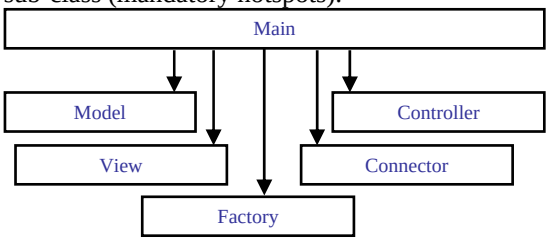

Figure 3: REMF Architecture-oriented Recipes

Given that the resolution of most of the hotspots was delayed to source code development time, we say that the instantiation followed a *code-centric* approach as opposed to a *design-centric* one (driven mainly by design model manipulations).

Although the decisions made were able to reduce the complexity of the REMF instantiation process the level of abstraction was still too low, as it was driven by architectural elements, following a code-centric approach, and lacking a more appropriate tool support.

Applying our Approach

After carefully understanding and documenting how previous REMF instantiations occurred we started our case study. Our goal was to produce a new application instance based on the REMF architecture. In particular, we aimed at developing a real-time monitoring tool for a credit card authorization system (named *WatCreditCard*). As we did not have an actual credit card server application and it was not relevant for the purpose of our case study, we specified stub classes that would produce random data to represent credit-card transactions (ex.: purchases, cancellations, credits, etc.). The steps we followed to construct the new instance are detailed next and as expected used our approach to framework instantiation.

The REMF UML Model- Since our approach encourages a design-centric development paradigm the very first step was to make sure that the REMF UML model was current. Therefore, decided to perform a reverse engineering process to obtain the revised design. Following this step, we exported the REMF UML class diagram to a file conforming with the XMI (XML Metadata Interchange) format, that would later serve as an input to our framework instantiation tool (xFIT).

Creating the RDL Script- In the next step, we created the RDL script that would guide the REMF instantiation process. We started by creating an RDL Pattern Library to represent REMF design patterns, in particular, the patterns used to expose hotspots. For each pattern we defined a corresponding RDL Pattern. Five design patterns have been implemented: Factory Method, Template Method, Strategy, Observer, and Abstract Factory.

Next, we created the REMF RDL script. For each recipe in the REMF cookbook we specified an equivalent RDL recipe with the appropriate parameters. As a result we ended up with six recipes: *Model*, *View*, *Controller*, *Connector*, *Factory* and the *main* recipe (Figure 3).

However, as the previous REMF adaptations had already revealed the level of abstraction of the instantiation process had not been appropriate, disclosing too many details of the framework architecture, especially its MVC-based design. Since now we had a supporting tool to cope with the low-level instantiation steps we decide to raise the level of abstraction during the instantiation process by hidden low-level details from the application developers. We rearranged the recipes in the REMF cookbook document in order to move from an *architecture-oriented* perspective to a *feature-oriented* one (see Figure 4).

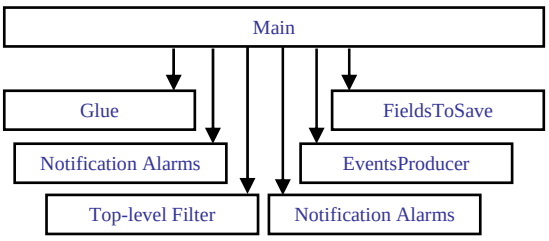

Figure 4: REMF Feature-based Recipes

For example, *Model*, *View*, and *Controller* recipes were substituted by others such as *NotificationAlarms*, *ApplicationFilters*, and *EventsProducer*. A feature-oriented recipe embraced REMF hotspots represented by design elements that may crosscut different architectural modules. Therefore, at the same time the level of abstraction was raised, since architectural details were omitted, the need for automation became essential in order to properly cope with the inherent crosscutting nature of the design elements handled. Thus, the provision of a supporting tool for framework turned out to be critical in our approach. After restructuring the REMF cookbook document we created the corresponding REMF RDL script.

Next, we depict the REMF RDL script we used to obtain a new framework instance in our case study. Some details were omitted for space reasons.

```
COOKBOOK REMF
RECIPE main
  modelClass, viewClass, controllerClass, connectorClass :
CLASS;
  -- Instantiating a Factory to create MVC Components
  CALL_RECIPE( Glue, (modelClass, viewClass, controller-
Class, connectorClass) );
  -- Handling Notification Alarms
  CALL_RECIPE( NotificationAlarms, (viewClass) );
  -- Handling Application-Specific Filters
  CALL_RECIPE( AppFilters, (viewClass) );
  -- Specifying the Application Top-Level Filter
  CALL_RECIPE( TopLevelFilter, (controllerClass) );
  -- Specifying the Events' Fields to be Saved
  CALL_RECIPE( FieldsToSave, (modelClass) );
  -- Specifying the Application Message Producer
  CALL_RECIPE( EventsProducer, (controllerClass, model-
Class) );
END_RECIPE

RECIPE NotificationAlarms ( IN viewClass : CLASS )
  concreteAlarmClass : CLASS;
  -- Hotspot HNA1,  Pattern: Factory Method, Creating an
Alarm Object, Type: Optional
  CALL_PATTERN( FactoryMethod, ( "AbstractView", "cre-
ateAlarm", viewClass ) );
  -- Hotspot HNA2, Pattern: Strategy,  Specifying the Alarm
Action,  Type: Optional
  CALL_PATTERN( Strategy, ( "AbstractAlarm", "raise",
concreteAlarmClass ) );
END_RECIPE

RECIPE AppFilters( IN viewClass : CLASS )
     ...
END_RECIPE

RECIPE TopLevelFilter( IN controllerClass : CLASS )
     ...
END_RECIPE

RECIPE FiltersToSave( IN modelClass : CLASS )
     ...
END_RECIPE

RECIPE EventsProducer( IN controllerClass: CLASS , IN:
modelClass : CLASS)
     ...
END_RECIPE

RECIPE Glue ( OUT modelClass, OUT viewClass, OUT con-
trollerClass, OUT connectorClass )
     …
END_RECIPE
END_COOKBOOK
```

Defining the Application Requirements- Before running the instantiation process we detailed the requirements of our new *WatCreditCard* application instance. This phase was especially important as it helped us identifying the optional hotspots to be adapted.

Table 9 depicts the *WatCreditCard* instance goal, its specific requirements, and the corresponding hotspots. As expected, the 10 mandatory hotspots had to be adapted since there was no default implementation provided for them. As for optional hotspots, 3 of them were redefined and 7 reused the framework default implementation.

WatCreditCard **Goals, Requirements and Hotspots**
Goal: Monitor real-time credit-card transactions produced by the *WatCreditCard* application server (ex.: purchase, cancellation, credit, terminal initialization, terminal shutdown). The application server was represented by *Stub* classes to produce random credit-card-related transactions.
R1: Raise audio alarms for events Hotspots Involved: • Creating an Alarm Object (*optional/adapted*) • Specifying the Alarm Action (*optional/adapted*)
R2: Expunge mal-formed transactions Hotspots Involved: • Defining the Application Top-Level Filter (*optional/adapted*)
R3: Save all transaction's information Hotspots Involved: • Specifying the Event Msg's Fields to Save (*optional/default*)
R4: Enable visual filtering for Credit-card transactions Hotspots Involved: • Handling Visual Filter Selection Changes (*mandator*) • Creating a Visual Filter Panel (*mandatory)*

• Creating a Textual Filter Expression (*mandatory)*
R5:Produce Credit-card real-time transactions
Hotspots Involved:
• Specifying the Events Message's Fields Meta Definitions (*mandatory*)
• Preparing Producer Resources (*optional/default*)
• Releasing Producer Resources (*optional/default*)
• Producing New Events (*mandatory*)
• Defining a New Message Producer (*mandatory*)
• Specifying the Appropriate Action when Producer Starts Producing Messages (*optional/default*)
• Specifying the Appropriate Action when Producer Stops Producing Messages (*optional/default*)
• Specifying the Appropriate Action when Producer is Repairing an Error in a Msg. Prod. (*optional/default*)
• Specifying the Appropriate Action when Producer Produces a New Message (*optional/default*)

Table 9: *WatCreditCard* Requirements/Hotspots

Producing Application Increments- In order to produce the WatCreditCard instance we ran our framework instantiation tool xFIT. xFIT took as input the REMF model files (XMI) and the REMF RDL script. During the instantiation process the application developer was required to answer instantiation questions (e.g. names of classes and methods, and whether or not an optional hotspot should be adapted). At the end, xFIT produced a UML model (in XMI) containing the framework and the application instance design elements. Not only classes, as occurred in previous REMF adaptations, but also methods were properly extended. Because of xFIT active guidance all hotspots were properly addressed and reviewed by the application developer.

Completing the Stubs Produced- Once the UML model was produced for the WatCreditCard instance, we performed a code generation process using our case tool in order to generate stub classes for the WatCreditCard instance. After adding specific code, and compiling/debugging our application was finally deployed.

4 Discussion

In this section we discuss how our approach improved framework instantiation and describe some future extensions in the light of our case study. In short, we claim that our approach i) raises the level of abstraction of framework adaptation processes, ii) enforces correctness of instantiation tasks, iii) provides adequate means for representing and reusing framework instantiation activities, and iv) reduce application development time.

4.1 Process Improvements

Correctness on Design Manipulation- As we have mentioned, previous REMF adaptations were performed based solely on the REMF cookbook, leaving it up to the application developer to understand and perform the instantiation steps. Thus, the instantiation process became time-consuming and error-prone. For example, method and class extensions were normally achieved through multiple design manipulation. Figure 5 below illustrates a typical manual method extension scenario in which the *raise method* has been mistakenly misspelled in the sub-class (*step 4*).

Not surprisingly, it was quite common for application developers to make mistakes such as misspelling methods/classes names, setting up erroneous inheritance relationships, specifying method signatures with the wrong type or number of parameters, and so forth. That is why in previous adaptations of the REMF application developers were encouraged to postpone method extensions to a later phase, typically at source code manipulation, when the compiler would be helpful in assist with correctness issues.

Let us examine a practical example. Looking at our case study consider the recipe *EventsProducer*. Lots of instantiation steps were provided towards the construction of *Event Producers*. Whenever a mandatory step was mistakenly performed (ex.: the *getMetaDefinitions* method signature in the sub-class did not mach with the super-class one) the application instance produced contained an error. This problem could be considerably more catastrophic if we think about a large framework containing numerous hotspots.

Another example of lack of support for design correctness regards the need to establish correct connections among major REMF architectural elements. The advantageous use of design patterns in the REMF architecture also required that design elements played specific roles and interacted with each other in a particular way. In order to conform properly with a pattern's specifications, application developers had to through manual means to ensure i) make sure that design elements have been correctly created and assigned to their corresponding pattern roles, and ii) ensure that the con-

nections among the design elements were properly set.

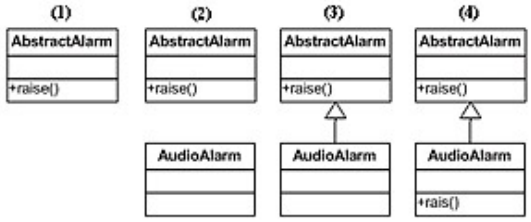

Figure 5: Manual method overriding steps

In contrast, our case study proved useful in supporting correctness of instantiation activities. RDL scripts provided the right mechanisms to ensure correctness based on syntax precise manipulation. For example, the RDL method extension command ensured that the overridden methods in the super-classes would always have the exact same signature in the sub-classes. Therefore, method and class extensions were always performed correctly in our case-study.

RDL Patterns were also important mechanisms to enforce design correctness. By properly combining RDL instantiation commands, RDL Patterns were able to connect design elements correctly into micro-architectures removing that burden from the application developers. For instance, the RDL recipe *Glue* called the RDL Pattern *AbstractFactory* in order to create a *Concrete Factory* and a set of *Concrete Products* (Model, View, and Controller) for the *WatCreditCard* application instance. The RDL pattern hid from the application developer the complexity of assembling the design elements needed to conform to the Abstract Factory design pattern. Likewise, the *Template Method*, whose instantiation involved a lot of method extensions, was also implemented in terms of RDL patterns decreasing the chances for errors.

By providing correctness support at the design level our approach also provided a means to reduce application construction time/cost once it was understood that the instantiation activities performed always handled the design elements in a correct way.

Effective Handling of Optional Hotspots- A difficulty reported in previous adaptations of the REMF regarded the effective handling of optional hotspots. As the instantiation process did not provide active guidance developers were required to find the proper design elements associated with the optional hotspots and correctly extend them. However, as the design elements may be spread throughout different framework architecture modules it turned out to be a time-consuming task for application developers to extend the optional hotspots properly. In addition, the consequences of missing a hotspot could be as serious as producing an application with undesirable features and thus having to repeat the whole instantiation process again.

In contrast, our approach provided effective mechanisms to cope with optional hotspots. RDL scripts were able to capture and represent all REMF optional extensions and present them progressively to application developers as the instantiation process was performed. Then, based on the specific application requirements application developers were able to decide which hotspots should be extended.

This explicit guidance through optional extensions kept the instantiation decisions under the control of the application developer, avoided runtime application behavioural problems, and shorten ed the application construction process.

Reuse of Framework Instantiation Artefacts- Some important artefacts can be reused across different framework instantiations. For example, the framework architecture and the cookbook document can be reused across multiple adaptations of the same framework. However, these artefacts do not provide effective means to capture and represent the expertise of the framework team in describing the actions needed to produce a valid and correct application instance. Therefore, the advantageous knowledge of the framework developers is wasted thus compelling application developers to create their own instantiation strategies. Indeed, *knowledge waste* occurred with all REMF previous adaptation processes.

In contrast, our approach introduced two new artefacts to framework instantiation: the RDL scripts and the RDL Pattern Libraries. As we discussed, RDL scripts embodied detailed instantiation steps for a given framework and could be reused across distinct framework adaptations. Framework developers had an adequate means to represent their *know-how* and explicitly represent the instantiation steps required to produce valid framework's instances. For instance, the REMF RDL script devised in our case study specified a standard and safe way to customize the REMF

architecture thus discouraging the application developers to define their own strategies.

Similarly, RDL Patterns represented recurring instantiation tasks that may occur in different framework instantiation processes. Moreover, they produced reusable RDL Pattern Libraries for the cases in which the patterns described were domain independent (e.g. Gamma patterns [1]). Thus, RDL Patterns were also important reusable artefacts that were able to communicate effectively the framework developer's expertise across different framework projects.

Feature versus Architecture-oriented Guidance- As we have mentioned, some important factors influenced the way the recipes were organized in the REMF cookbook document for the first five adaptations. Since there was no assisted guidance or tool support for framework instantiation the manipulation of design model elements corresponding to the framework hotspot was complex and may have involved addressing various architectural modules. In addition, time-to-market requirements of some of the REMF application instances compelled us to specify a more pragmatic approach to facilitate framework instantiation. As a consequence, the REMF cookbook recipes were arranged to follow an *architecture-oriented approach*..

Architecture-oriented recipes usage was advantageous in this scenario especially because the design elements manipulated were normally found concentrated in a few architectural modules. Figure 6 depicts three architecture-oriented recipes *Model, View* and *Controller* that manipulates design elements within same-named framework modules. By ensuring that application developers did not need to cope with scattered design manipulations we kept the instantiation process more manageable. However, despite the efforts to improve the REMF instantiation process it remained error-prone, time-consuming and too low-level, because it revealed the intricacies of the framework architecture to the application developers.

In contrast, one of the main goals of our approach was to raise the level of abstraction of framework instantiation. Application features rather than architectural modules should guide the instantiation process. In fact, the RDL process language in conjunction with the supporting tool provided (xFIT) enabled a f*eature-oriented* arrangement of the recipes in the REMF cookbook. Architectural solutions were kept hidden from application developers as the same time as crosscut manipulations of design elements could be carried out in a straightforward manner. xFIT took care of handling multiple architectural modules spread throughout the REMF design model. Figure 6 shows *Feature-Oriented* recipes manipulating design elements that crosscut multiple REMF architectural modules (*Model, View* and *Controller*).

The REMF RDL script mapped each framework feature to its corresponding architectural components through recipes. In order to facilitate the understanding of the feature-oriented recipes in our approach we make an analogy between RDL recipes and Aspects in Aspect-Oriented Programming [14],[18]. While Aspects are normally used to modularize crosscutting concerns related to tangled and scattered code, RDL Recipes modularize *crosscutting instantiation tasks*.

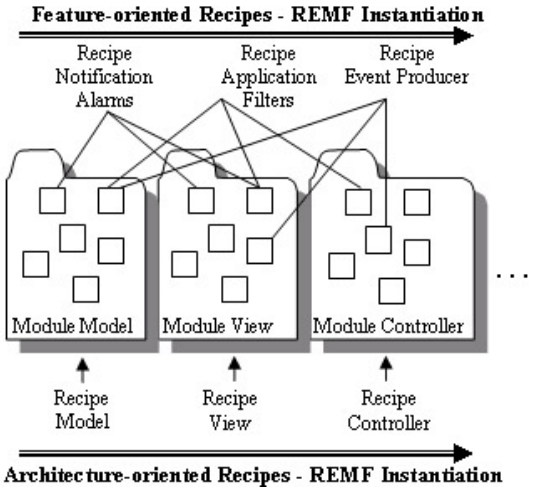

Figure 6: Feature x Architecture-based Recipes

Now, the advantages of using our approach became more evident. The level of abstraction of the instantiation process was raised from an architectural to a feature-oriented perspective, the steps were actively guided, and the time to produce new application instances shortened.

Design versus Code-centric Approach- Unassisted REMF previous instantiations tended to follow a *code-centric approach*. Although design manipulations had been encouraged most of the instantiation steps were postponed to the implementation phase (e.g. method extensions). This way we expected the Java compiler to work as a validation tool, especially in the case of mandatory hotspots (method overriding). Although the deci-

sions taken helped reduce the general complexity of the REMF instantiation process the level of abstraction turned out to be excessively low-level centered on source code manipulations and driven by architecture-oriented recipes.

On the other hand, our approach encourages a design-based instantiation process. RDL scripts manipulates and transforms UML class diagrams in such a way that source code details are hidden and postponed for a future phase. After obtaining the final UML class diagram for the application instance ,application developers are encouraged to complete the corresponding source code. Doubtless, a design-centric approach is much more convenient once it allows application developers to progressively cope with instantiation details. Basically, it offers two phases: i) adapting framework design to obtain an application instance(design), and ii) completing source code stubs (implementation).

4.2 Extensions to Our Approach

Although the application of our approach in the REMF case study presented very interesting results in terms of automation/guidance, support for design correctness, and shortening of development time, we were also able to identify some possible extensions.

First, the lack of integration between our approach and a framework documentation approach, especially in terms of tool support, was missing. Changes in REMF cookbook recipes are not reflected on RDL recipes and vice-versa. As well, we had to consult the REMF cookbook at times during the instantiation process in order to provide the right inputs to the xFIT tool.

Second, only UML class diagrams were handled by our approach. It was our intention to enhance the REMF UML documentation by exploiting new UML diagrams but we realized there would be no gain in terms of the framework instantiation.

Finally, our approach did not handle source code generation in any way. For example, in our case study just the design elements of the *WatCreditCard* instance were produced and the source code generation relied on the Case tool we had chosen.

5 Related Work

Several approaches claim to facilitate framework instantiation. In [9] the authors used a structured specification to support framework instantiation. Although their work led the area of instantiation assistance, their contribution can be summarized as a template to a reuse document expressed in natural language, which can be hard to follow due to the lack of a formal construction. Moreover, a Cookbook can not be processed automatically, living space for inconsistencies in its definition. As an extension to Cookbooks, Hooks [7] also provided a template for framework instantiation assistance and they share the same problems.

In the area of instantiation guidance Smartbooks [15] advocate the use of Software Agents to execute instantiation plans. The main issue with this approach is the introduction of non-standard notations, such as TOON [15], which causes an extra burden to the Framework Development. In OBS [10] the authors used a generative approach to framework instantiation that shares some characteristics with our previous work [3]. However, the OBS approach is based on ready-to-use black-box frameworks, which constraints instantiation processes to component configuration not customization. FRED [29] is a framework editor tool that uses specialization patterns in order to generate applications. However, FRED is code-centric and tied to the Java language whereas our approach is design-centric and programming-language independent. Finally, UMLAUT [32] presents a general UML transformation framework based on algebraic compositions and reified elementary transformation. Our approach also performs transformations on UML models but is rather focused on the domain of framework instantiation.

6 Conclusion

In this paper, we presented our approach to framework instantiation, the enhancements we have made to improve its effectiveness, and a case study we developed to evaluate our approach. Our case study showed that the use of our approach was helpful in raising the level of abstraction of framework adaptation processes. Our approach encouraged a feature-oriented design-centric framework instantiation process. Moreover, It effectively supported correctness by assisting de-

sign manipulations and providing active guidance throughout the adaptation process. We also identified some extensions needed to improve our approach as our research moves further on.

As for future work, we aim to extend our approach to support the instantiation of Aspect-Oriented Frameworks [11],[19] since we believe that the use of aspects as means to separate software concerns is a trend. Furthermore, we want our approach to support other UML diagrams, initially Interaction and Activity diagrams. Finally, the idea of using generative techniques [31],[10] to enable our approach to go beyond design manipulations is a potential target for investigation.

Acknowledgements

This work has been partially supported by the Natural Sciences and Engineering Research Council (NSERC) and CAPES (Brazilian Ministry of Education Agency).

About the Authors

Marcilio Mendonca received the MSc degree in computer science (1996) from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil. He is currently a PhD student in the School of Computer Science at the University of Waterloo, Canada. Prior to the PhD he worked for 10 years in industry and academia as a Software Architect, Project Manager, and University Lecturer.

Paulo Alencar is a Research Professor in the School of Computer Science at the University of Waterloo. Dr. Alencar has received international research awards from organizations such as Compaq (Compaq Award for best research paper in 1995 for his work on a theory of evolving software systems), and IBM (IBM Innovation Award, 2003). His research, teaching, and consulting activities have been directed to software engineering in general and his current research interests specifically include software design, architecture, composition, Web-based and hypermedia systems, software processes and formal methods. His work on software engineering has been recognized by NSERC international referees as “clearly excellent,” “very accomplished,” and of “great value.” His last joint NSERC Strategic Project Grant was considered as being in the top-three in Canada. He has been the principal or co-principal investigator in many national and international projects supported by NSERC, CITO, Precarn, IBM, Bell, Rogers, Sybase, and many other software companies and funding agencies in Canada, Germany, Argentina, and Brazil.

Toacy Oliveira Toacy C. Oliveira received the BSc degree in electrical engineering (1991) and the MSc (1997) and PhD (2001) degrees in computer science from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil. He also spent two years as a postdoctoral fellow at the University of Waterloo, Canada. He is currently professor at the University of Liverpool, UK and at Pontifical Catholic University of Rio Grande do Sul, Brazil. Dr. Oliveira has participated in several projects in cooperation with industry and worked as a consultant to the United Nations Development Programs (UNDP). His current research interests include software design, software processes and tools.

Don Cowan is Distinguished Professor Emeritus in the School of Computer Science at the University of Waterloo. He was the funding Chair of Computer Science at the University of Waterloo and is currently Director of the Computer Systems Group at the same University. His current research interests include software engineering, software tools, Web-based systems for asset management, software processes, and hypermedia documentation. Dr. Cowan is the designer of twenty unique Web-based information portals.

References

[1] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, ***Design Patterns, Elements of Reusable Object-Oriented Software***, Addison-Wesley, 1995.

[2] Oliveira, T. C. Filho, I. M., Lucena, C. J. P. , Alencar, P. S. C., Cowan, D. D., ***Software Process Representation and Analysis for Framework Representation***, IEEE Transactions on Software Engineering, March 2004, Volume 30, Issue 3, p.145-159.

[3] Oliveira, T.C., Alencar, P., Cowan, D.: ***Towards a declarative approach to framework instantiation,*** Proceedings of the 1st Workshop on Declarative Meta-Programming (DMP-2002), September 2002,Edinburgh, Scotland, p 5-9.

[4] Fontoura M., Pree W., Rumpe B., ***The UML Profile for Framework Architectures,*** Addison Wesley, 2001.

[5] Ortigosa A., Campo M., ***SmartBooks: A Step Beyond Active-Cookbooks to Aid in Framework Instantiation***,Technology of Object-Oriented Languages and Systems, 1999.

[6] Fayad M.E., Schimdt, D.C, Johnson R., ***Application Frameworks.*** In Fayad, M.E., Schimdt, D.C., Johnson R. (Eds.), Building Application Frameworks-Object-Oriented Foundations of Frameworks Design. John Wiley, New York, 1999.

[7] Froehlich G., Hoover H., Liu L., Sorenson P., ***Hooking into Object-Oriented Application Framework***, ICSE'97, IEEE Press, 491-501, 1997.

[8] Pree W., Pomberger G., Schapert A., Sommerlad P., ***Active Guidance of Framework Development***, Software-Concepts and Tools (1995) 16: 94-103, Springer-Verlag.

[9] R. Johnson, ***Documenting Frameworks Using Patterns***, OOPSLA'92, ACM Press, 1992, p. 63-76.

[10] Vaclav Cechticky, Philippe Chevalley, Alessandro Pasetti, Walter Schaufelberger, ***A Generative Approach to Framework Instantiation***, Lecture Notes in Computer Science, Volume 2830, Nov 2003, Pages 267 – 286.

[11] Rausch A., Rumpe B., Hoogendoorn L., ***Aspect-Oriented Framework Modeling***, AOSD International Conference 2003, Workshop on Aspect Oriented Modeling.

[12] ***Struts Project***, The Jakarta Software Foundation, http://struts.apache.org/

[13] ***Omondo EclipseUML Project***, www.omondo.com/

[14] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., and Irwing, J., ***Aspect-Oriented Programming***, Procedings of ECOOP'97, Springer Verlag, pages 220-242, 1997.

[15] Ortigosa A., Campo M., Salomon R., ***Towards Agent-Oriented Assistance for Framework Instantiation***. In Proc. OOPSLA '00, Minneapolis, Minnesota USA, ACM SIGPLANNotices, 35, 10, 2000, 253-263

[16] ***The Eclipse Project***, IBM, http://www.eclipse.org/

[17] ***The Unified Modelling Language***, OMG, http://www.uml.org/

[18] ***Aspect-Oriented Software Development,*** AOSD.NET, http://aosd.net/

[19] Constantinides C., Bader A., Elrad T., Fayad M., and Netinant P., ***Designing an Aspect-Oriented Framework in an Object-Oriented Environment***, ACM Computing Surveys, 32(1), 2000.

[20] ***Jade-Java Agent Development Framework***, Jade Research Group, http://jade.tilab.com/

[21] ***Java Technology***, Sun Microsystems, http://java.sun.com/

[22] ***JUnit Testing Framework***, JUnit.org, http://www.junit.org/

[23] ***CORBA - Common Object Request Broker***, OMG, http://www.corba.org/

[24] Weinand A., Gamma E., Marty R., ***ET++ - An Object-Oriented Application Framework in C++***. In OOPSLA'88, Special Issue on SIGPLAN Notices, 23(11), 1988.

[25] Wilson D.A., Roseinstein L.S., Shafer D., ***Programming with MacApp***, Reading, Massachusetts: Addison Wesley, 1990.

[26] Feuer A., ***MFC Programming***, Addison Wesley, 1997.

[27] ***JBoss-AS Webpage***, JBoss, http://www.jboss.org/

[28] Krasner G, Pope S., ***A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80***, Journal of Object-Oriented Programming, 1(3):26-49, 1988.

[29] Hakala M., Hautamaki J., Koskimies K., Paakki J., Viljamaa A., ***Annotating Reusable Software Architectures with Specialization Patterns***, Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01) - Volume 00, 2001

[30] ***The HotDraw Framework***, st-www.cs.uiuc.edu/users/brant/HotDraw

[31] Czarnecki K., Eisenecker U., ***Generative Programming-Methods, Tools and Applications***, Addison Wesley, 2000.

[32] Ho W.M. , Jezequel J., Guennec A.L., Pennaneac'h F., ***UMLAUT: an Extendible UML Transformation Framework***, INRIA, Research Report #3775, Oct/1999.