RESEARCH ABSTRACT

TITLE: Computational Support for the Evolution of OO Frameworks

RESEARCH AREA OR SUB-AREA: Software Evolution

TECHNICAL PROBLEM

The high costs of developing software motivate the reuse and evolution of existing software. Object oriented frameworks constitute one of the most promisory techniques to reuse both design and code.

A framework is an abstract design that defines a software architecture that may be reused for a specific domain problem. This architecture is defined in terms of collaboration contracts between classes and by a set of variation points, which define where the framework can be customized.

Good frameworks are usually the result of many iterations and a lot of hard work involving structural changes. These iterations make the relationships among classes more explicit and interfaces more abstract. In this way, a framework constitutes an ever-evolving representation in terms of variations and commonalities as consequence of diverse factors related to domain changes. However, as frameworks evolve and their interfaces change, the advantages of using a framework are lessened. Not only the developers are responsible for performing normal maintenance activities, but they also must evolve their programs in accordance with the evolution of the framework. This adds overhead to using the framework.

The most complex problem regarding evolution and iterating development of frameworks is the impact of the changes in the framework design on the rest of the system and possible incompatibility with previous customizations. The Figure 1 illustrate this problem, in which the domain changes represents the new requirements where the M() method semantics differs from the one previously supported by the framework. As a result, the application tends to drift away from the framework architecture.
Another problems with framework evolution are described in [1] and summarized below:

- Structural complexity: framework evolution can make its structure (object interfaces, class hierarchies, and so on) hard to manage and understand;
- Changes in the domain: as the framework evolves and new framework instances are created, new abstractions that should be part of the framework may be derived;
- New design insights: the framework’s design structure may need to be improved in the light of issues previously neglected or forgotten.

Actually, there is a very little support for framework evolution. The lack of appropriate methods and techniques to support the redesign and evolution makes the framework development expensive and difficult.

**Prior Research**

Currently there are very few framework design methods that deal with framework evolution. A pattern-based description of some accepted approaches underlying framework design can be found in [2].

Refactoring rules have already been successfully used to assist framework maintenance [3, 4]. However, refactorings are behavior-preserving transformations and cannot, for example, deal with problems related to changes in the framework domain. The unification rules, proposed in this work, are complementary to the refactoring procedures and address some problems not supported by them.

The Refactoring Browser [5] is a tool to help maintainence of framework written in Smalltalk. It currently does not support unification rules, but it has an open architecture and the introduction of unification and new refactoring procedures seems to be straightforward. The design pattern tool proposed in [6] also uses refatorings to archive framework restructuring.

Roberts and Johnson propose the development of concrete applications before actually developing the framework itself [2] they claim that framework abstractions can be derived from concrete applications. The unification-based development process may be used to systematize this approach.
Ideally, the behavior preservation of the unification processes should be proven formally. In practice and in previous research [7], this generally has not been done.

**PROPOSED SOLUTION**

In this work we propose the use of refactorings [3, 4] and unification rules [7], to assist framework maintenance and evolution.

Refactorings are changes made to the internal structure of software that improve the framework design to make it easier to understand and cheaper to modify, preserving the semantics of the original design. Unification rules are special refactorings that both preserve and extend the framework semantics. Rules are especially useful to incorporate new features into the design. Sometimes, the application of refactoring procedures before the unification can improve its result. In this way, the combination of refactoring and unification rules may be useful to support framework evolution.

Framework adaptation normally takes place by completing the variation points defined by its architecture. However, there are many cases in which the framework does not support the required customization and the application developers need to violate its structure. This phenomenon is referred as architectural drift: the intended framework architecture and the architecture that underlies the current implementation of a given framework instance become different [1]. Unification rules can be used to avoid this phenomenon, by making the necessary transformations in the framework structure to incorporate the changes required by a given framework instance and by the restructuring of framework’s variation points during evolution. The term *unification* is used to indicate that the rules are applied to “unify” the framework architecture with the architecture of the violating instance. After the application of unification transformation the set of applications that may be instantiated based on the framework is enlarged, since new variation points are defined.

In summary, refactorings and unification rules preserve the behavior of a program in computational sense, in order to avoid the impact of the changes in the framework design on the rest of the system and customizations already existent. This implies that these transformation processes always result in legal programs equivalent to the original program. In particular, rules extend the semantics of the framework design by adding the semantics of new features, increasing the set of applications that may be instantiated from it.

**CONTRIBUTIONS**

The major contributions of this research are:
- It identifies a set of program restructurings to apply on object-oriented frameworks.
- It shows how to semi-automatically support refactorings and unification processes in a way that preserves the behavior of a program.
- It defines in detail the evolution processes.
• A formal specification to illustrate the semantics of the transformation process. This formal specification allows us to reason about process properties through the model checking techniques to check structural evolution properties.

• A process-based approach to specify the evolution processes, and the constraints that should hold when the framework evolve.

• Techniques to support the verification of the framework architecture and evolution processes in order to analyze through Prolog proofs and model checking.

METHODS

This work describes an approach for providing automated support for the restructuring and evolution of object-oriented frameworks using refactorings and unification rules. The solution is based on the formal description of the framework semantics and evolution processes.

The system that supports the verification of structural and behavioral properties of the design is implemented as a combination of two complementary techniques: logic programming and model checking.

The process of model checking is to determine whether a system specification possesses a certain property. A finite model of the system is constructed by performing an exhaustive state space search. After the model has been successfully constructed in Prolog and compiled by the model checker, we can verify the system properties or query the design. The properties are usually expressed as temporal logic formulas.

In the initial phase, the framework designs are represented in Prolog and stored in a XSB Prolog database. The structural UML-based representation of the framework elements is specified in XSB Prolog [8] in terms of object-oriented design primitives in a predicate-like format. The framework evolution processes are translated to XL (a highly expressive process language extension of value-passing CCS) and the behavioral properties are specified in \( \mu \)-calculus.

For the analysis, the structural framework properties are verified using the XSB Prolog deductive facilities. This can be accomplished with the use of XMC [9, 10], a model checker for verifying the behavioral properties expressed in \( \mu \)-calculus temporal logic.

REFERENCES


