A Rule-Based Approach to Framework Evolution

Mariela Inés Cortés
mariela@inf.puc-rio.br

Marcus Felipe Montenegro Carvalho da Fontoura
IBM Almaden Research Center
San Jose, CA, USA
fontoura@almaden.ibm.com

Carlos José Pereira de Lucena
lucena@inf.puc-rio.br

Puc-RioInf. MCC 44/03 October, 2003

Abstract. Framework development is very expensive, not only because of the intrinsic difficulty related to capturing the domain knowledge, but also because of the lack of appropriate methods and techniques to support the evolution of the framework architecture. In this paper, we introduce the concept of evolution rules and describe its role in the context of framework development. Evolution rules allow the restructure and the addition of new features into the framework design, making sure that these changes are consistent with the applications previously instantiated. There are two kinds of evolution rules: refactorings and extensions. We describe both of them, showing how they can be combined to provide a useful support to framework evolution. In addition, we propose a methodology to prove the correctness of evolution processes. This methodology is based on CCS formalism and model checking techniques. The evolution approach is illustrated through Avestruz, a framework for web searching.

Keywords: framework evolution, refactoring rules, extension rules.

Resumo. O desenvolvimento de *frameworks* é caro, não só pela dificuldade intrínseca relacionada à captura do conhecimento do domínio, mas também por causa da falta de métodos e técnicas apropriados para dar suporte à evolução da sua arquitetura. Neste trabalho é introduzido o conceito de regras de evolução descrevendo o seu papel no contexto de desenvolvimento de *frameworks.* As regras de evolução permitem restruturar e adicionar novas caracteristicas no *design* garantindo que essas mudanças são consistentes com as aplicações previamente instanciadas. Existem duas classes de regras de evolução: *refactorings* e extensões. Ambas são descritas apresentando como podem ser combinadas para dar apoio à evolução de *frameworks.* Também é proposta uma metodologia para provar a corretude dos processos de evolução. Esta metodologia é baseada no formalismo CCS e técnicas de verificação de modelo. A abordagem proposta é ilustrada através de Avestruz, um *framework* para busca na web.

Palavras-chave: evolução de *frameworks*, regras de *refactoring*, regras de extensão.

1. Introduction

A framework[1] is an extensible semi-finished piece of software that represents a generic solution to a set of applications in a specific domain. A framework is composed of a *kernel* subsystem, which is common to all the applications that may be generated within the framework, and *variation points*, which implement application specific behavior [6]. Therefore, the framework behavior is strongly bound to the set of applications that may be instantiated from it.

A framework constitutes an ever-evolving representation of our knowledge of the domain in terms of variations and commonalties. The lack of appropriate methods and techniques to support its evolution makes framework development expensive and difficult. The most complex problem regarding framework evolution is the impact of the changes in the framework design on the rest of the system, and possible incompatibility with previously defined applications.

Given the well-known complexity and iterative nature of object-oriented framework development, the basic philosophy described in [2, 5] may be summarized by:

Framework development = Framework evolution

In this paper we introduce the concept of evolution rules describing their role in the framework development process. There are two kinds of evolution rules, refactorings and extensions. The main contribution of this paper is to show that:

Framework evolution = Framework refactoring + Framework extension

Refactoring of source code [15, 19] is a well-known approach suggested for the development and evolution of frameworks by restructuring a program in the way that it allows other changes to be made more easily. Several refactorings incorporate design patterns into the code [16]. The application of one of these patterns expresses the intended use and adaptation possibilities in a higher-level of abstraction encapsulated by the pattern semantics. In specific situations, refactorings also implement extensions into the framework design, creating variation points [10]. In this case, the original behavior of the framework is transformed in an alternative behavior that is encapsulated into a prefabricated class into the framework design.

Extension rules [7, 8, 12] are based on metapatterns [21], which are the basic principles of object-oriented software construction using a combination of template-hook methods. Several design patterns are based on metapatterns [26]. Thus, the extension step is carried out more efficiently through extension rules since a larger number of situations can be modeled, including those shaped through refactorings. These rules can be viewed as *meta-refactorings* that modify the variation point structure to support the incorporation of an alternative behavior into the

[1] We use the terms framework and product line synonymously.

framework design. Complicated changes to a program can require both refactorings and extensions.

The combination of refactoring and extension rules to evolve framework designs in a controlled way is the key point of this work. These technologies are very useful in developing efficient and flexible application frameworks and they fit well into the iterative framework development process. Both refactoring and extension rules preserve the observable behavior of the original design[2]. We also present a formal verification of the correctness of the evolution process based on refactorings and extension rules on the basis on syntactic and semantic analyses. The syntactic analysis is trivial because it can be made through the language compiler and metrics. The semantic analysis requires the behavioral equivalence between the program representation before and after the evolution process. The approach proposed in this work to establish the behavioral equivalence is based on abstract interpretation theory and model checking techniques.

The remainder of this paper is organized as follows: Section 2 describes the semantics of refactorings and extension rules. Section 3 presents the methodology proposed for the formal verification of evolution rules. Section 4 illustrates how refactorings and extension rules can be applied in practice, using the Avestruz framework as an example. In Section 5 we comment on some related works. Finally, Section 6 presents our conclusions and future research directions.

2. Evolution Rules

In [4], framework development is considered equivalent to the evolution process involving the execution of two tasks: restructure and extension. In this work we propose the use of evolution rules to support the execution of these tasks. The rules are based on two complementary approaches: refactoring [15, 19] and extension rules [7, 8, 12]. Both techniques are used to restructure the code and to add new abstractions, with the framework architecture becoming more flexible and, therefore, promoting its reutilization. Evolution rules model changes that may be applied to a system, preserving the original behavior in the computational sense.

[2] Please note that evolution rules preserve the behavior of a program in computational sense. This implies that these transformations always result in legal programs equivalent to the original program.

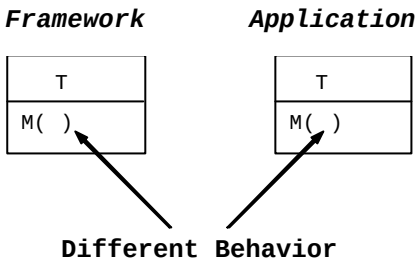

Figure 1. Architectural drift problem

Evolution rules may be used to avoid the architectural drift problem [6] by changing the variation point structure of the system. This phenomenon occurs when the framework does not support the required customization and the application developers need to violate the framework structure. Figure 1 illustrates this problem, in which the domain changes represent 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. When the design modeled by the application is not a valid instance of a framework, evolution rules may be applied to add flexibility into the framework design. The framework flexibility is based on the variation point structure that is accessed by the users (application developers). In this way, evolution rules can be considered behavior-extending transformations, since the cardinality of the application set is increased after the evolution process.

2.1. Refactoring

Refactoring is the activity of redesign of a program unit to take advantage of programming techniques, such as design patterns, to improve the design. Refactoring is a technique that also plays a major role in extreme programming [24] and can occur at various times throughout the development process. In this section, several refactorings that support programming and evolution activities are illustrated.

2.1.1 Refactorings to support code reorganization

The transformation represented in Figure 2 represents the *Push Up Method* refactoring that moves a common behavior to the abstract superclass [15] to avoid the duplicated code problem. In this example, since the same method M() belongs to the classes A and B, a new superclass is created and the redundant method is moved.

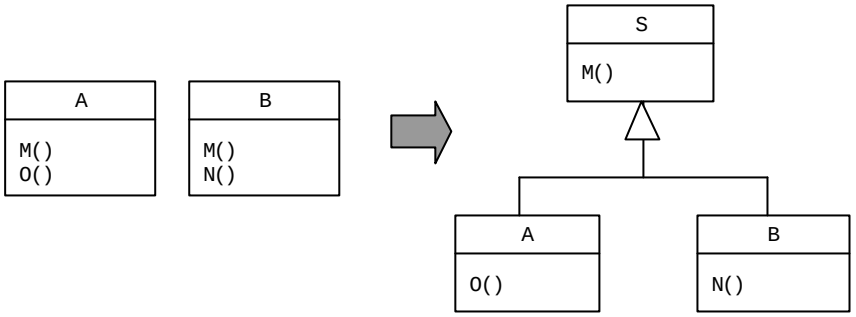

Figure 2. Application of the refactoring *Push Up Method*

In the resultant class hierarchy the common behavior is concentrated in the superclass. Meanwhile, specific and additional behavior is added to the subclasses, avoiding code redundancy. Based on the new structure (Figure 3), new variant behavior can easily be modeled by the addition a variation points to the superclass.

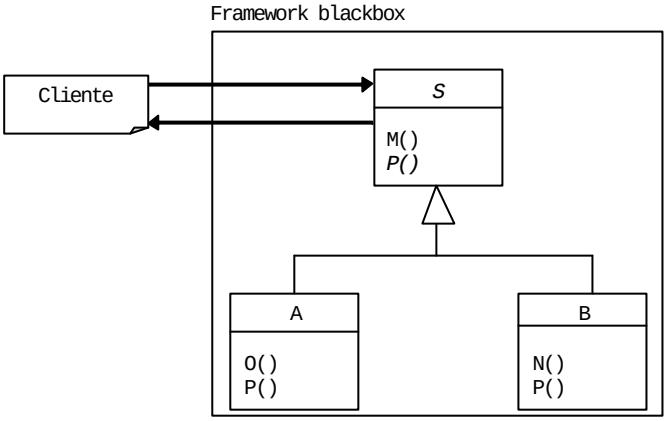

Figure 3. Creation of a blackbox variation point subsystem

The structure represents the design of a blackbox framework consisting of a variation points subsystem that offer two alternative behaviors for method P() implemented in the concrete subclasses A and B.

A large part of Fowler´s refactorings [15] describes methods to package code properly. The key refactoring in this category is *Extract Method* (Figure 4), which takes a code fragment and turns it into its own method. Subsequently, a call to the new method is used to replace the removed fragment.

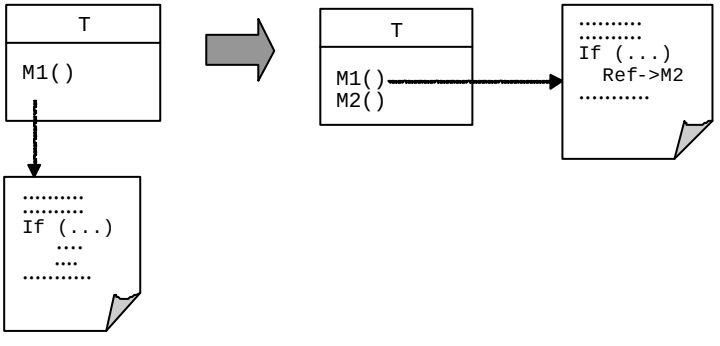

Figure 4. *Extract Method* refactoring [15]

Extract Method changes the system structure through the incorporation of the *Unification* metapattern [21] into the design, where the old method M1() is transformed into a template method, that invokes the new (hook) method M2(). As it will be shown later, this refactoring is used in the mechanics of several extension rules.

2.1.2 Refactorings to support design patterns

A natural relation between patterns and refactorings is presented in the design patterns catalogue by Gamma et al. [16]: "Patterns... supplies targets for your refactorings". In other words, refactorings allows designers to focus on basic patterns when they are developing software projects. Patterns can be added through refactorings: "… refactorings turn explicit the design patterns that are subjacent into the code" [16]. The use of design patterns has costs related to complexity and indirection. For this reason, design should be as flexible as needed, not as flexible as possible.

Refactorings have been shown to directly implement certain design patterns [26]. Examples of refactorings with this property are *Replace Type Code with State/Strategy* and *Form Template Method* [15]. In the rest of this section we present some of these refactorings[3].

Replace Type Code with State/Strategy (Figure 5) implements directly the transformation that incorporates the *State/Strategy* pattern into the design. This refactoring substitutes the code type of a generic object with a state object, adding one subclass for each type.

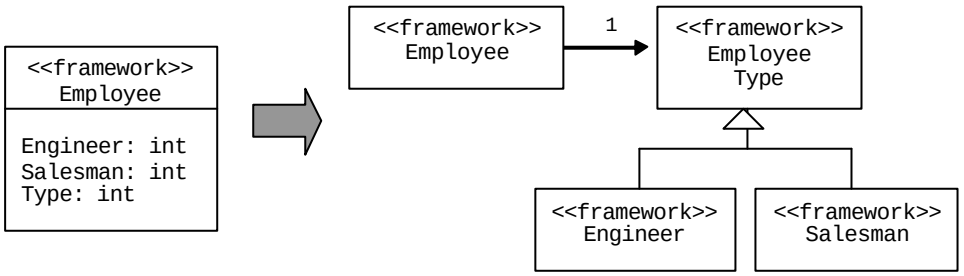

Figure 5. *Replace Type Code with State/Strategy* refactoring [15]

In this way, the variability that before was manipulated through mechanisms as switch-case features, are manipulated by a class hierarchy in the target design. *Replace Conditional with Polymorphism* [15] describes a similar situation. Both of these refactorings are equivalent to the extension rule *Add Unification Pattern* (compare Figure 8 in Section 2.2.2 with Figure 5 to see the similarities).

[3] Note that when refactorings are applied on frameworks, the structure of variation points may be modified, affecting the set of custom applications.

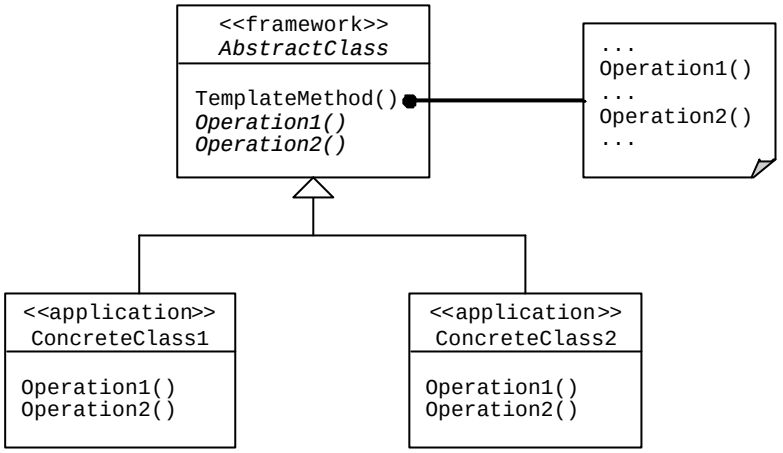

Figure 6. Pattern *Template Method* [16]

The second example presents the class structure of the *Template Method* pattern (Figure 6). This pattern defines a skeleton of behavior in a method template, which can be tailored to provide different behavior through hook methods in subclasses. This pattern is based on the *Unification* metapattern [14].

Form Template Method is a refactoring that incorporates the *Template Method* pattern into the design. This refactoring can be applied when two methods in subclasses (the class hierarchy already exists) execute similar steps in the same order, yet the steps are different. In the target design, the common code is factored in the superclass and the variant behavior is implemented in the subclasses, preventing the duplication.

2.2. Extension Rules

Extension rules can be viewed as *meta-refactorings* used to extend the framework behavior, making it possible to instantiate a greater number of applications. Evolution rules are transformations that alter the variation point structure of the framework. They are based on framework metapatterns [21], which implement variation points as a combination of template and hook methods [16, 20, 21]. A *template method*[4] provides the skeleton of a behavior. A *hook method* is called by the template method and can be tailored to provide different behaviors. There are four evolution rules, which automate the incorporation of the basic framework patterns proposed by Pree [21]. In the remainder of this section we present each of them by means of a set of short descriptions including:

- The description of the associated metapattern;
- The motivation to describe why the rule should be implemented;

[4] Template method must not be confused with the C++ template construct , which has a completely different meaning.

- The solution proposed;
- The mechanics of how to carry out the evolution.

The evolution processes are illustrated in terms of UML class diagrams. We use visual representations for the UML-F tags *framework* and *application* [14].

2.2.1 Add Hook Rule

Description. This rule is used to incorporate a hook method into the design.

Motivation. The framework architecture does not support the required customization because a method of the kernel subsystem is not able to realize the behavior required for the application developer.

Solution. The instance application must change the implementation of a kernel method. In this way, each application can define alternative behaviors. Figure 7 illustrates this process.

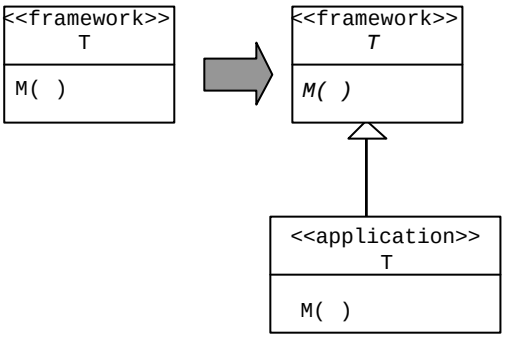

Figure 7. Application of Add Hook rule

Mechanics. The mechanics can be resumed in the following steps:

1. Create a subclass for each instance application, if it does not already exist.
2. Create subclass methods that override the original methods.
3. Make the superclass method abstract.

2.2.2 Add Unification Pattern Rule

Description. This rule is used to incorporate the *Unification* pattern into the design. This pattern occurs when both the template and hook methods belong to the same class.

Motivation. Application developers need to add flexibility to existing methods. The framework architecture does not support the required customization because the behavior supplied by the kernel methods is not completely adequate to the behavior required for the application developer. This might happen in any of the following situations:

- New insights in the domain. Some application specific concepts may become general concepts and must be incorporated into the framework;

- New design insights. Some design issues that were neglected in the framework's initial design phase are discovered and need to be incorporated into the framework kernel.

Solution. Implementing the variant steps as a combination of template-hook methods. Create a hook method, which executes the special behavior required by the application developer. The method might be created through the *Extract Method* refactoring [15], which replaces a fragment of code with a call to the newly created method (Figure 8).

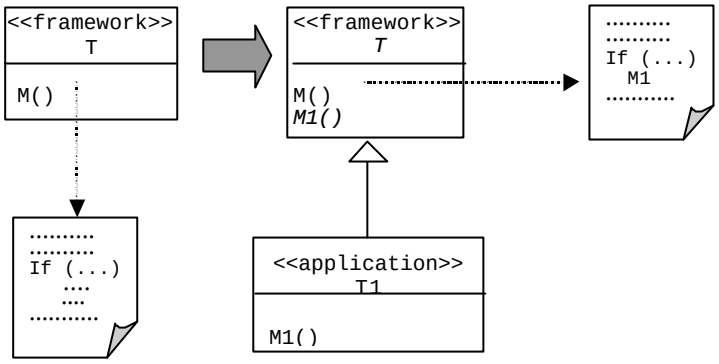

Figure 8. Application of *Add Unification Pattern*

Mechanics. The mechanics may be summarized in the following steps:

1. Create the new method using the *Extract Method* refactoring [15].
2. For each application, if it does not already exist, create a new subclass.
3. Move the implementation of the new method to each subclass using the *Push Down Method* refactoring [15].
4. Make the superclass method abstract.

2.2.3 Add Separation Pattern Rule

Description. This rule is used to incorporate the *Separation* pattern into the design. This pattern occurs when the template and hook methods belong to different classes.

Motivation. The motivation here is the same as the one in *Add Unification Pattern*¸ but in this case the final design is more flexible – since the template and the hooks belong to different classes, adaptations can happen during runtime.

Solution. Create a new variation point method in a separate hook class (Figure 9). This variation point must be extended by composition. In the obtained design in addition to the variation point subsystem an additional class is required to host the template method.

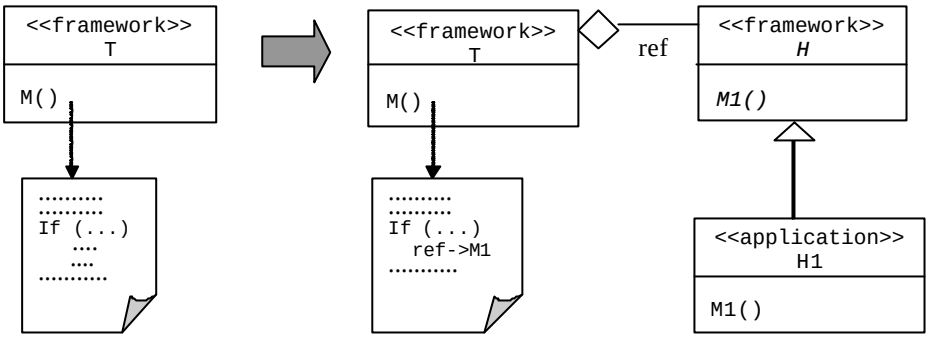

Figure 9. Add Separation Pattern rule

Mechanics. The evolution process in this case may be summarized in the following steps:

1. Create the new method using the *Extract Method* refactoring [15].
2. Create a new class using the *Extract Class* refactoring [15].
3. Create a subclass for each instance application, if it does not already exist.
4. Make the superclass abstract.

2.2.4 Add Recursive Pattern Rule

Description. This rule is used to incorporate the *Recursive* pattern into the design. This pattern occurs when an object of the template class refers objects of its hook class. The template class is a descendent of the hook class.

Motivation. Sometimes, it may be useful to create object compositions. These compositions can be treated as a simple object, which are useful to modify the object behavior without modifying the class structure already existent.

Solution. Create an object composition to allow handling object collections in order to selectively add or modify behavior to instances (Figure 10).

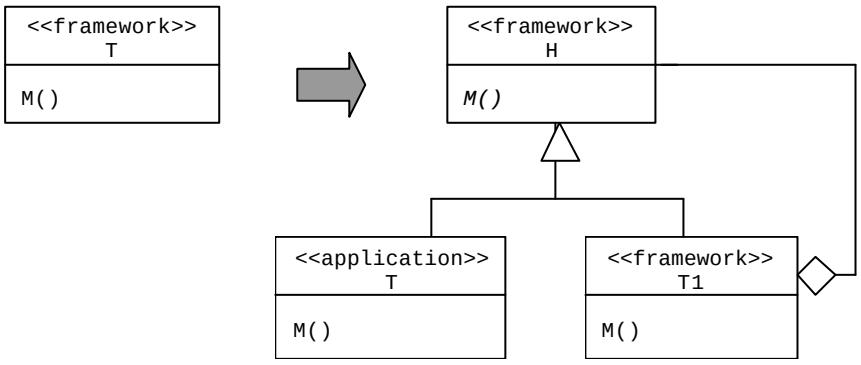

Figure 10. Application of *Add Recursive Pattern* rule

In the design based on the *Recursive* metapattern any number of template classes can be defined as subclasses of H. These template classes can define additional/modified behavior. Note that any number and combination of instances of template classes can be attached to instances of H descendants.

During execution time, the hook class H forwards the control to either descendant (T or T1). The trick behind this design is that T1 can modify the behavior of methods and forward the call to the object referred by the reference. Furthermore, instance variables and methods can be added. This means that the modified behavior implemented in the composite class T1 can be attached to all descendants of H. If the call is forwarded to a T object, the behavior is preserved; if the call is forwarded to a T1 object, the behavior is extended with the associated special behavior.

Mechanics. The application of this evolution rule may be summarized in the following steps:

1. Create the composite class.
2. Create the abstract superclass.
3. Add an instance variable of the composite class in the hook class.

2.3 Comparing Refactorings and Extension rules

Refactorings can be very useful to support framework restructuring and extension on the basis of specific design patterns. However, two points differentiating refactorings and evolution rules must be highlighted:

- There are no refactorings for all design patterns (for example, no refactoring addresses the incorporation of design patterns that are based on recursive subsystems). This incompleteness in the refactoring catalogue [15] can be solved through the use of extension rules, which are based on metapatterns. Metapatterns model all the possible combinations of template and hooks methods, including the recursive composition. Consequently, more situations can be described.
- Some refactorings support the creation of blackbox variation point subsystems. This process does not change the framework behavior since it simply implements restructuring and moving of code already existing in the framework. In this way, the variant behavior that already exists inside the framework becomes explicit in the design by the use of refactorings. This does not change the framework nor make any new concrete classes. Differently, extension rules always incorporate new variation points in the system.

Design patterns are specific instances of the metapatterns, which are the simplest construction principles. Since extension rules implement the introduction of metapatterns into the design, design patterns may be introduced using the same mechanics. Figure 11 describes some patterns that may be introduced using refactorings and extension rules. Each pattern is associated to the correspondent higher level metapattern.

Metapattern	Design Pattern	Evolution Rules	Refactoring Rules

Separation	Builder	Add Separation	---------
Unification	Factory Method	Add Unification	Replace Constructor With Factory Method
Separation	Prototype	Add Separation	---------
Separation	Bridge	Add Separation	---------
1: n Recursive	Composite	Recursive	---------
1:1 Recursive	Decorator	Recursive	---------
1:1 Recursive	Chain of Responsibility	Recursive	---------
Separation	State / Strategy	Add Separation	Replace Type Code with State / Strategy
Unification	Template Method	Add Unification	Form Template Method

Figure 11 Metapatterns, patterns, extension rules and refactorings

All referred patterns can be introduced through the extension rules, including those that can be generated using specific refactorings. Moreover, the applicability of extension rules is extended to a larger set of different situations than refactorings. For example, refactorings do not support the introduction of patterns based on recursive structures.

3. Formal verification

Ideally, the behavior preservation of refactorings should be proven formally [26]. In practice and in previous research, this generally has not been done. Instead of formal proofs, the approach originally proposed by Barnejee and Kim for database schema evolution has been adopted [3]. In [19] a set of seven invariants to preserve behavior for refactorings was proposed, but no proof that these invariants preserved program behavior was presented.

In this paper we propose a formal methodology to verify the correctness of the evolution rules (refactoring and extension rules). In this sense, the system behavior, both before and after the evolution process, is formally defined. Based on these formal descriptions, equivalence relations are used to check if the two programs may be considered equivalents. The methodology uses CCS [17] as the formalism for the description of the program behavior and model checking techniques [9] to establish the behavioral equivalence. The methodology proposes the following steps for the formal behavior verification:

1. Create the state diagrams on the basis of the state of variables used by the program, before and after the program transformation.

   a. The diagram denotes the sequence of states. Each state denotes a particular situation of the variables manipulated by the program. To facilitate this step the statements in the program are labeled.
2. Add the transitions that determine the control sequence throw in the program execution to the diagrams. The transitions may be classified into two groups:
   a. Named. These transitions connect different states in the diagrams. These transitions are denoted with different symbols in the diagrams.
   b. Invisibles. These transitions connect equivalents states (initial and final) in the diagrams. These transitions are denoted with the same symbol in diagrams, usually the t (*tau*) symbol.
3. Translate the diagrams into a CCS specification [17].
4. The two formal specifications are incorporated in the CWB (*Edinburgh Concurrency Workbench*) model checking tool. CWB [9] is an interactive system that allows the user participation throw commands to check properties and to make analyses. The command **eq** is used to check the behavioral equivalence between processes. The system returns *true* if the behaviors of the process may be considered equivalents in the observational sense, and *false* in another case.

In following, we apply the proposed methodology to verify the correctness of a refactoring process. In this very simple example we illustrate the wrong use of the *Rename Method* refactoring. The original code with the labels and the corresponding sequence of states is presented in the Figure 12.

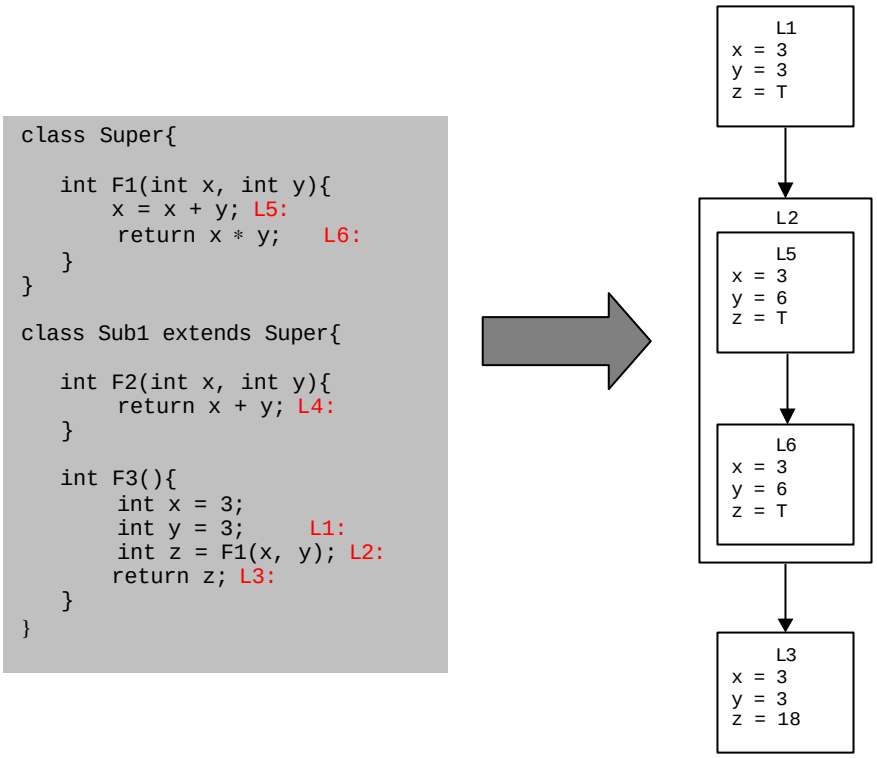

Figure 12. Original code and state sequence before the refactoring process

The refactoring *Rename Method* is applied to rename the function F2 to F1. Note that a method with same name and parameters already exists in the class Super. Thus, the behavior of the modified code possibly was modified. The code after the evolution process is presented in Figure 13.

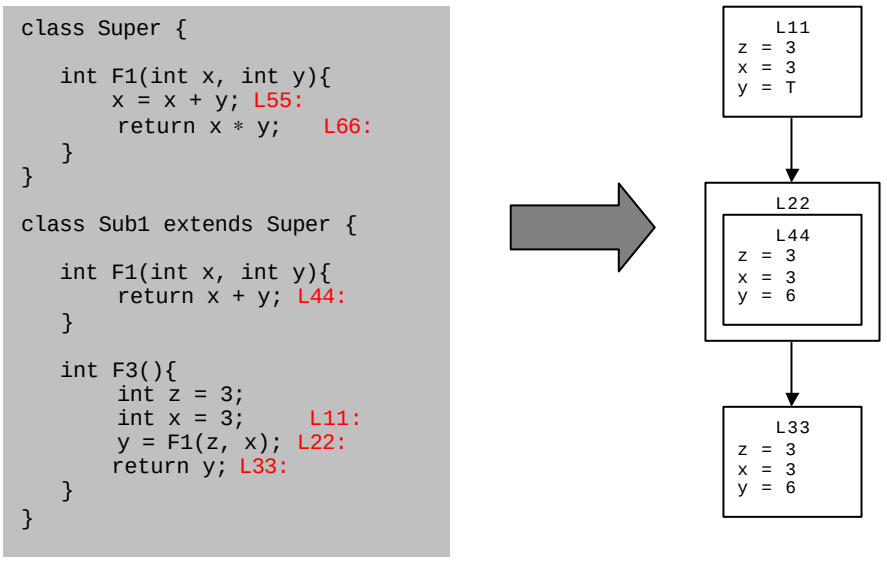

Figure 13. Transformed code and state sequence after the refactoring process

In the next step, the transitions in both sequences are classified as named or invisibles. The resultant diagrams of transitions are depicted in the Figure 14. In this case all transitions are named.

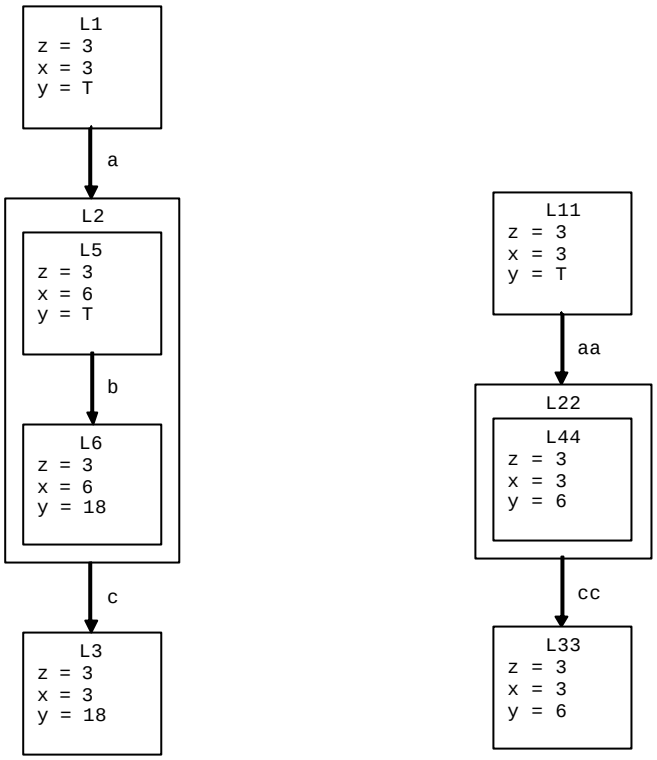

Figure 14. Transition diagrams before and after the refactoring process

Figure 15 presents the CCS description of the transition diagrams described in above. This description is used as input for the CWB model checking tool. In the CCS formalism each state in the diagram is modeled using the **agent** abstraction and the observational equivalence relation between the descriptions is determined with the **eq** command.

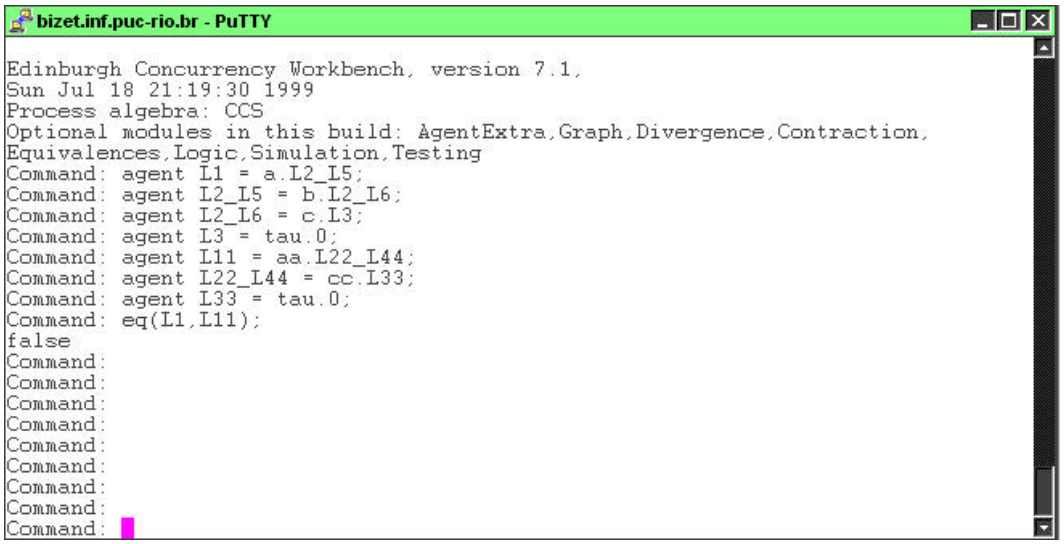

Figure 15. CWB Session for the *Rename Method* refactoring

The system return *false* and the session conclude. In this way, on the basis of the methodology proposed, is possible determine that the application of the *Rename Method* refactoring in this case is not behavior preserving.

4. Study Case: Avestruz framework

The proposed approach is illustrated through the Avestruz framework, which is an ongoing project at the TecComm Laboratory (PUC-Rio) (http://www.teccomm.les.inf.puc-rio.br). Avestruz is a search engine framework designed to process and to parse HTML pages. The processing determines whether particular words are localized in the text of the pages. In the crawling process, links are extracted and stored for future searching. This is done in parallel, using several machines. The initial architecture of the framework is the simplest one that allows accomplishing these requirements.

The core of the system is the ClScanner class. It is responsible for localizing the URL of the initial page of the site and transferring the control to the HTML processor module to parse the page. The abstract class ClHTMLDoc defines a variation point to allow independence between the framework and the HTML parser used in the custom applications. Figure 16 shows the class model and the semantics of the initial framework design.

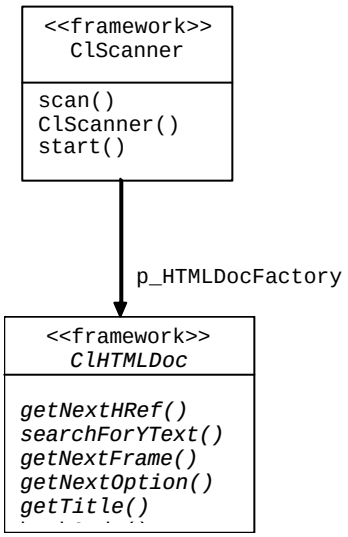

Figure 16. Searching engine framework

When the framework finishes parsing a page it must recover the next page to be visited using the set of links that was extracted in the parsing process. During crawling, the index structures are stored by the ClScanner class according to the order of appearance in the HTML documents. Later, during runtime, applications need more flexibility and efficiency for recovering the search results. Then, we use evolution rules to create a new variation point in the framework that allows the customization of the strategy for storing the index information.

We use *Add Separation Pattern* to create a new entity decoupled from ClScanner class, allowing customizations during runtime. Figure 17 illustrates a new diagram and semantics for the Avestruz modules, with the changes generated by the evolution transformation. During this

evolution process, the methods of the ClScanner class are transformed into template methods and their specific behavior is translated to the ClList class.

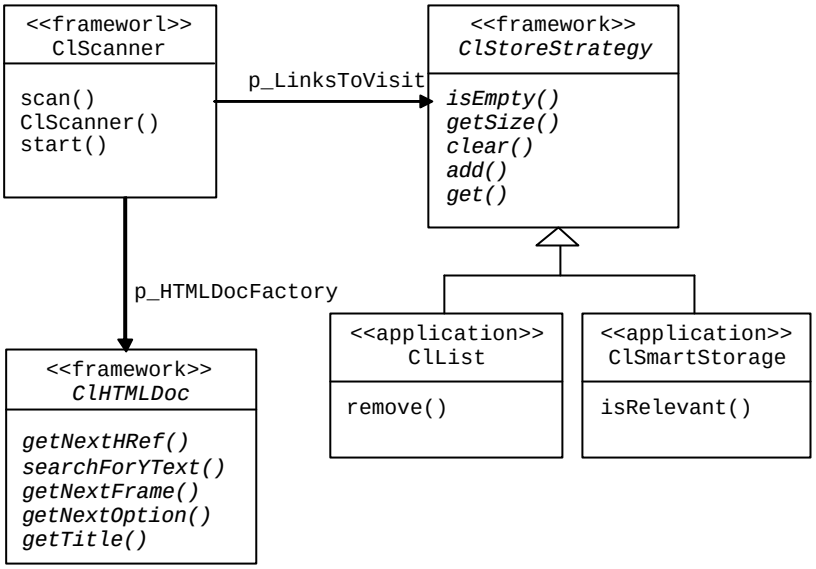

Figure 17. Adding flexibility for the link processor module

In the context of the custom application, the transformation is behavior-preserving since the structure of the ClScanner and ClList classes is suitable as the original design. On the other hand, the framework behavior is extended because a new structure of template and hooks methods was created during the evolution process. In the current design, overriding the hook methods in the ClStoreStrategy class can modify the behavior of the template class ClScanner.

In its initial design, the *ClStoreStrategy* class was specialized to use lists. Later, a new strategy was developed for storing the page links, ClSmartStorage, which implements a more intelligent strategy that determines whether a link is relevant for the current search. Thus, the search is restricted to a specific site, avoiding searching in pages outside of the current scope.

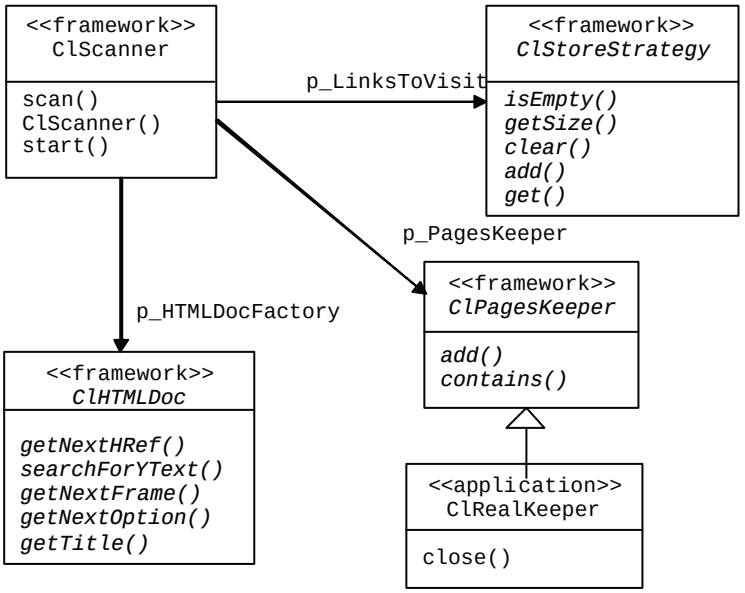

Figure 18. Adding a permanent memory into the design

Frequently, a web site includes repeated pages within a collection of related pages. When the framework was tested for different applications, in several situations the best strategy was to ignore the repeated pages. For that, a new abstraction was required to model a permanent memory of the visited pages, adding new requirements to the design. Consequently, we use evolution rules to add this evolution to the framework. In this case, we use *Add Separation Pattern* rule to create a new class decoupled from ClScanner, *ClPagesKeeper*, to retain the history of the visited pages. The result for this transformation is shown in Figure 18.

The semantics of the scan() method in ClScanner class is modified by the application of the evolution rule because the specific behavior that implements the memory of visited pages is extracted and moved to ClRealKeeper class. If no memory is needed, the situation may be modeled with the new application class ClDummyKeeper.

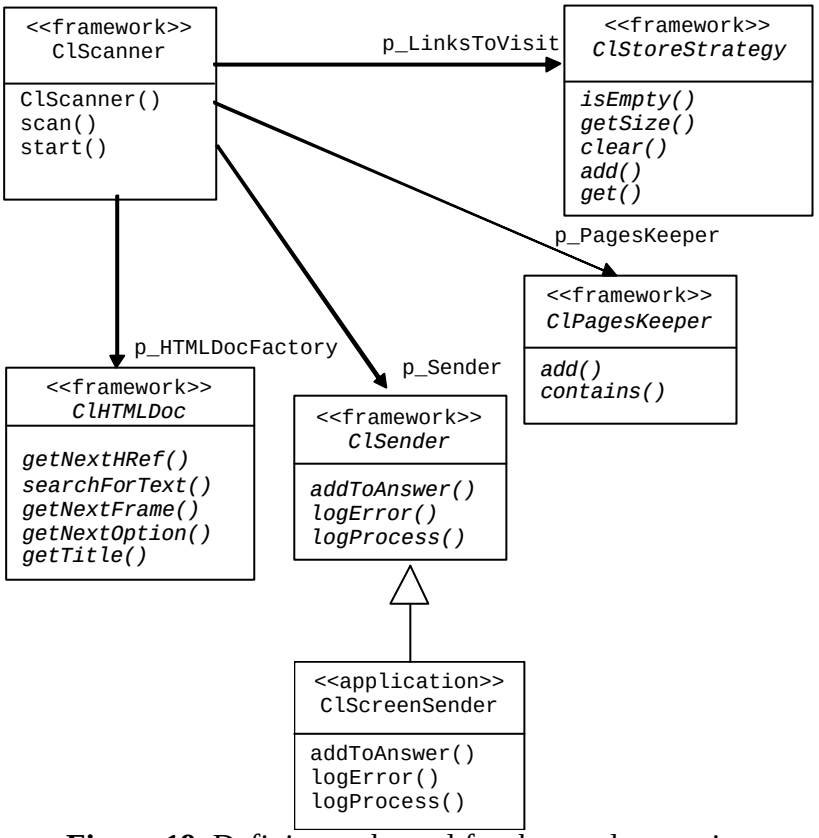

Figure 19. Defining a channel for the result reporting

When the search of relevant pages is finished, the search results must be reported to the user. In the initial solution, the results were returned through a DOS window (System.out). To improve the design and to avoid the mixing of the result reporting with the parsing code, we create a new entity representing the channel that carries out the result reporting. This step of evolution is a behavior-preserving transformation because it improves the framework design but does not add semantics and does not modify the variation point structure. Consequently, refactorings come into play to support this evolution step. The most suitable refactoring for this situation is *Extract*

Class [15]. In consequence, the ClScreenSender class is extracted from ClScanner class to show the results via DOS window. In this way, the original class is transformed in two classes with clear responsibilities. After the refactoring is done the semantics of the system is preserved since the behavior of ClScanner and ClScreenSender together is the same as the original design.

Usually, the custom applications require the return of formatted results that may be received via e-mail when the search is finished, or via browser in processing time. Considering the design shown in Figure 18, variations in the channel implementation may be required by framework instances. Then, evolution rules can be used to turn the channel into a variation point. This can be done by the application of *Add Unification Pattern* to create the abstract class ClSender, which allows the definition of specific channels in the custom applications (Figure 19).

Now, the new framework design allows the customization of several variation points, however, the answer processor is still insufficient in some situations. In particular, the ClScanner class is defined to accept only a ClSender object. It implies that the report of the searching results is realized through only one channel. However, in several situations reporting through more than one channel may be useful, e.g., mail and browser simultaneously.

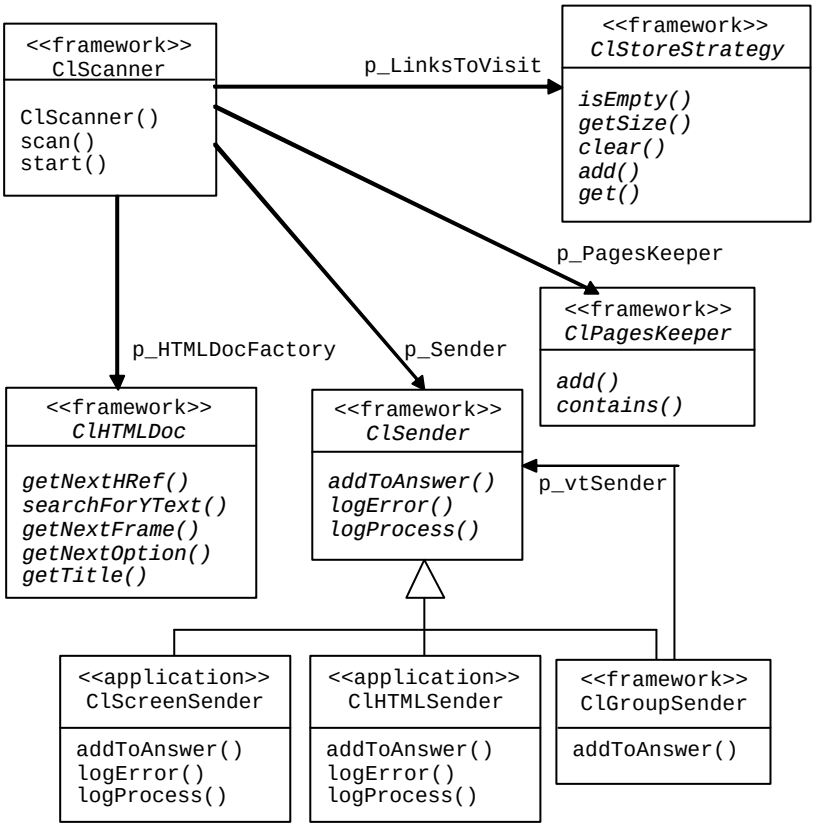

Figure 20. Evolution to report results throws several channels

To reach this flexibility, new semantics must be added into the design. Therefore, unification processes are used to describe this step of evolution. The more suitable rule is *Add Recursive*

Pattern, because it allows creation of a group of channels used by the custom application to the result reporting, which are represented by the ClGroupSender class. Figure 20 represents the final design of the Avestruz framework after the unification process.

5. Related Work

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 [22]. Some interesting aspects regarding framework design such as framework integration, version control and over-featuring can be found in [6].

The Refactoring Browser [23] is a tool to help maintenance of frameworks 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 [11] also uses refactorings to achieve framework restructuring.

Roberts and Johnson propose the development of concrete applications before actually developing the framework itself [22]. They claim that framework abstractions can be derived from concrete applications. The unification-based development process may be used to systematize this approach. An approach that integrates framework and XP is presented in [24].

A model for framework development based on viewpoints is proposed in [13]. This method was used as our first approach to framework design, and the current version has been refined through the development of several case studies.

In relation to the formal aspect, a theory for the formalization of the semantics of framework, based on the set of application-instance that can be generated, is presented in [12]. This semantics is defined using sets theory and meaning functions. However, this theory does not permit the establishment of property semantics in relation the behavior.

In [19], the processes of refactoring are verified in accordance with a set of program properties, derivatives of the database area, added to a property that refers to equivalence semantics between operations. However, this latter property is not formally verified.

6. Conclusions

This paper shows how evolution rules can be combined with refactoring rules to support framework maintenance and evolution. The applicability of evolution rules is analyzed in different stages of the evolution process. Evolution rules are transformations used to avoid the architectural drift problem [6] by restructuring framework variation points during evolution.

After the application of evolution transformations, the set of applications that may be instantiated based on the framework is enlarged, since new variation points are defined.

The paper also proposes a methodology for the formal verification of the correctness of evolution rules. The methodology is based on the CCS formalism to specify the behavior of programs. This formal specification will allow us to think about process properties using model-checking techniques, for example, to establish the behavioral equivalence of programs. This methodology, in conjunction with structural verification (i.e., syntactic analyses, metrics, etc.), can be very useful for understanding the evolution processes and their consequences on the framework design.

The ongoing work consist in the extension of an already existing tool for software refactoring also supporting extension rules and the elaboration of new rules for different kinds of variation points.

Acknowledgments

This work is being supported in part by the National Research Council of Brazil (CNPq).

References

1. Alencar P., Cowan D., Oliveira T., Lucena C. Process-Based Representation and Analysis of Frameworks Instantiations. In submission. July, 2001.
2. Batory D. et al. Construction of file management systems from software components. In Proceedings of COMPSAC. 1989.
3. Banerjee J, Kim W. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In: Proceedings of the ACM SIGMOD Conference. 1987.
4. Butler G., Xu L. Cascated Refactoring for Framework Evolution. Proceedings of 2001 Symposium on Software Reusability. ACM Press, pages 51-57, 2001.
5. Coad P. Object-oriented patterns. *Communications of the ACM,* v.35, n.9, p. 152-159, September 1992.
6. Codenie W., Hondt K., Steyaert P., and Vercammen A. From Custom Applications to Domain-Specific Frameworks. *Communications of the ACM,* 40(10), 71-77, 1997.

7. Cortés, M. Computational Support for the Evolution of Object Oriented Frameworks. In: Doctoral Simposium of the International Conference of Software Engineering. Orlando, Florida. May 2002.

8. Cortés M., Fontoura M., Lucena C. Using Refactoring and Evolution Rules to Assist Framework Evolution". To appear in Software Engineering Notes, vol 28 No 6, November 2003 Page 29. Available from http://www.teccomm.les.inf.puc-rio.br/publicacoes.htm.

9. CWB Model Checking. Available in: http://www.dcs.ed.ac.uk/home/cwb/.

10. Fayad, M. Application Frameworks. In: Building Application Frameworks. New York: Wiley, 1999. p. 3-28.

11. Florijin G., Meijers M, van Winsen P. Tool Support for Object-Oriented Patterns, ECOOP'97, LNCS 1241, Springer-Verlag, 472-495, 1997.

12. Fontoura M. A Systematic Approach to Framework Development. Ph.D. Dissertation, Computer Science Department, Pontifical Catholic University of Rio de Janeiro, 1999.

13. Fontoura M., Crespo S., Lucena C., Alencar P., Cowan D. Using Viewpoints to Derive Object-Oriented Frameworks: a Case Study in the Web-based Education Domain. Journal of Systems and Software, Elsevier Science, 54 (3), 239-257, 2000.

14. Fontoura M., Pree W., Rumpe B. *The UML Profile for Framework Architectures*, Addison-Wesley, 2001.

15. Fowler M. Refactoring: Improving the design of existing code. Addison-Wesley, 1999.

16. Gamma E., Helm R., Johnson R., and Vlissides J. Design patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

17. Milner, R. Communicating and concurrency. Prentice-Hall, Hemel Hempstead, 1989.

18. OMG, OMG Unified Modeling Language. Available in: http://www.omg.org.

19. Opdyke W., Refactoring Object-Oriented Frameworks, Ph.D. Dissertation, Computer Science Department, University of Illinois, Urbana-Champaign, 1992.

20. Pree W. Object-oriented versus conventional construction of user interface prototyping tools. Doctoral thesis, University of Linz, 1991.

21. Pree W. *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

22. Roberts D. and Johnson R. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks, in Pattern Languages of Program Design 3, Addison-Wesley, R. Martin, D. Riehle, and F. Buschmann (eds.), 471-486, 1997.

23. Roberts D., Brant J., and Johnson R. A Refactoring Tool for Smalltalk. University of Illinois at Urbana-Champaign, Department of Computer Science. Available in: http://st-www.cs.uiuc.edu/users/droberts/.

24. Roock S. eXtreme Frameworking - How to aim applications at evolving frameworks. Proceedings of the XP'2000 Conference, 2000.

25. Schmid, H. Design patterns for constructing the hot spots of a manufacturing framework. Journal of Object-Oriented Programming, v. 9, n.3, p. 325-37, jun. 1996.

26. Tokuda, L.; Batory, D. Evolving Object-Oriented Designs with Refactorings. Automated Software Engineering, v.8, p. 89-120, 2001.

27. Wirfs-Brock R., Wilkerson B., Wiener L. Surveying current research in object-oriented design. Communications of the ACM, 33(9), 1990.