Using UML-F to Enhance Framework Development: a Case Study in the Local Search Heuristics Domain

Marcus Fontoura♣, Carlos J. Lucena*, Alexandre Andreatta♦, Sérgio E. Carvalho*, and Celso C. Ribeiro*

♣ Department of Computer Science, Princeton University
35 Olden Street, Princeton, NJ 08544, U.S.A.
fontoura@cs.princeton.edu

* Computer Science Department, Catholic University of Rio de Janeiro
Rua Marquês de São Vicente 225, Rio de Janeiro 22453-900, Brazil
{lucena, sergio, celso}@inf.puc-rio.br

♦UNIRIO - University of Rio de Janeiro
Department of Applied Computer Science
Rua Voluntários da Pátria 107, Rio de Janeiro, RJ 22270, Brazil
andreatt@uniriotec.br

ABSTRACT

Currently frameworks are most commonly represented through design diagrams written in standard object-oriented analysis and design languages. However, these design notations do not provide elements for identifying framework variation points and how their instantiation should be performed. Therefore, in order to create framework instances, users have to rely on extra documentation that is not always available. This paper shows the benefits of an extension to UML that explicitly represents all the information required for framework development and instantiation. The approach is illustrated through a large real-world framework for local search heuristics for combinatorial optimization problems.

KEY WORDS: object-oriented frameworks, design notation, UML-F, framework development, framework instantiation.

1. INTRODUCTION

This paper illustrates the weakness of standard object-oriented design languages regarding framework documentation and shows how UML-F (Fontoura, 1999; Fontoura et. al., 2000) can be used to address this problem. It shows how the documentation of a large framework in the local search heuristics domain, namely the Searcher framework (Andreatta et. al., 1998), can be complemented through UML-F diagrams. This case study highlights the benefits of this approach regarding framework development and instantiation. Searcher was first described in (Andreatta et. al., 1998) through OMT diagrams (Rumbaugh et. al., 1994) and pattern descriptions (Gamma et. al., 1995). UML-F was not available at that time.

UML-F is a UML profile (D'Souza et. al., 1999) useful for modeling frameworks and product-line architectures. UML-F uses the basic UML extension mechanisms to define new constructs for modeling all the relevant aspects of framework designs. Several other projects have already been successfully developed with UML-F, as reported in (Fontoura, 1999). The Searcher case study is a condensed version of a real application of UML-F.

The rest of this paper is organized as follows. Section 2 describes the local search heuristics domain and presents the Searcher framework using the OMT design language (Rumbaugh et. al., 1994). Section 3 revisits the framework description using UML-F, and discusses the benefits of this new representation. Section 4 describes some related work. Finally, Section 5 presents our conclusions and future research directions.

2. LOCAL SEARCH HEURISTICS AND THE SEARCHER FRAMEWORK

Hard combinatorial optimization problems usually have to be solved by approximate methods. Constructive methods build up feasible solutions from scratch. Among them, we find the so-called greedy algorithms, based on a preliminary ordering of the solution elements according to their cost values. Basic local search methods are based on the evaluation of the neighborhood of successive improving solutions, until no further improvement is possible. As an attempt to escape from local optima, some methods allow controlled deterioration of the solutions in order to diversify the search (Andreatta et. al., 1998).

In the study of heuristics for combinatorial problems, it is often important to develop and compare, in a systematic way, different algorithms, strategies, and parameters for the same problem. The *Searcher* framework (Andreatta et. al., 1998) encapsulates different aspects involved in local search heuristics, such as algorithms for the construction of initial solutions, methods for neighborhood generation, and movement selection criteria. Encapsulation and abstraction promote unbiased comparisons between different heuristics, code reuse, and easy extensions.

2.1 SEARCHER PATTERN-BASED DESCRIPTION

This section describes the framework as it was first documented by its authors (Andreatta et. al., 1998), using a variation of the pattern form proposed in (Gamma et. al., 1995) and OMT diagrams (Rumbaugh et. al., 1994).

Intent:

To provide an architectural basis for the implementation and comparison of different local search strategies.

Motivation:

In the study of heuristics for combinatorial problems, it is important to develop and compare, in a systematic way, different heuristics for the same problem. It is frequently the case that the best strategy for a specific problem is not a good strategy for another. It follows that, for a given problem, it is often necessary to experiment with different heuristics, using different strategies and parameters.

By modeling the different concerns involved in local search in separate classes, and relating these classes in a framework, our ability to construct and compare heuristics is increased, independently of their implementations. Implementations can easily affect the performance of a new heuristic, for example due to programming language, compiler, and other platform aspects.

Applicability:

The *Searcher* framework can be used in situations involving:

- local search strategies that can use different methods for the construction of the initial solution, different neighborhood relations, or different movement selection criteria;
- construction algorithms that utilize subordinate local search heuristics; and
- local search heuristics with dynamic neighborhood models.

Structure:

Figure 1 shows the classes and relationships involved in the *Searcher* framework.

Participants:

- *Client:* contains the combinatorial problem instance to be solved, its initial data and the pre-processing methods to be applied. It also contains the data for creating the *SearchStrategy* that will be used to solve the problem. Generally, it can have methods for processing the solutions obtained by the *SearchStrategy*.
- *Solution:* encapsulates the representation of a solution for the combinatorial problem. It defines the interface the algorithms must use in order to construct and modify a solution. It delegates to *IncrementModel* or to *MovementModel* requests to modify the current solution.

- *SearchStrategy:* constructs and starts the *BuildStrategy* and the *LocalSearch* algorithms, also handling their intercommunication, in case it exists.
- *BuildStrategy:* encapsulates constructive algorithms in concrete subclasses. It investigates and eventually requests *Solution* for modifications in the current solution, based on an *IncrementModel*.
- *LocalSearch:* encapsulates local search algorithms in concrete subclasses. It investigates and eventually requests *Solution* for modifications in the current solution, based on a *MovementModel*.
- *Increment:* groups the necessary data for an atomic modification of the internal representation of a solution for constructive algorithms.
- *Movement:* groups the necessary data for an atomic modification of the internal representation of a solution for local search algorithms.
- *IncrementModel:* modifies a solution according to a *BuildStrategy* request.
- *MovementModel:* modifies a solution according to a *LocalSearch* request.

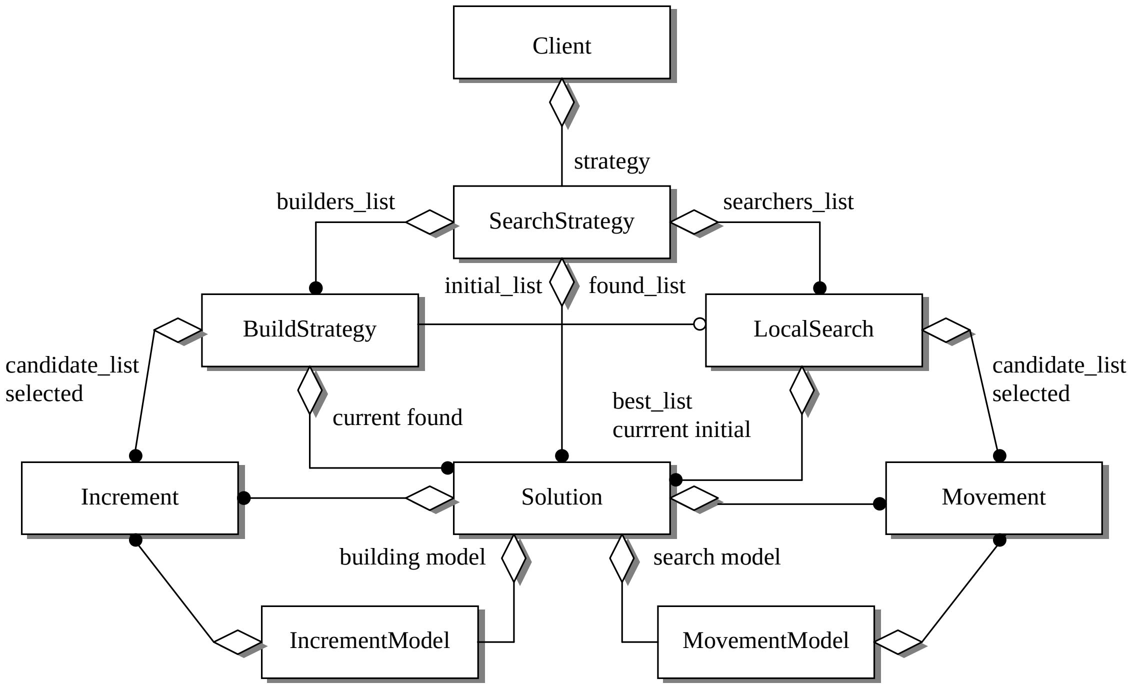

Figure 1. Searcher class diagram

Collaborations:

The *Client* wants a *Solution* for a problem instance. It delegates this task to its *SearchStrategy*, which is composed by at least one *BuildStrategy* and one *LocalSearch*. The *BuildStrategy* produces an initial *Solution* and the *LocalSearch* improves the initial *Solution* through successive movements. The *BuildStrategy* and the *LocalSearch* perform their tasks based on neighborhood relations provided by the *Client*.

The implementation of these neighborhoods is delegated by the *Solution* to its *IncrementModel* (related to the *BuildStrategy*) and to its *MovementModel* (related to the *LocalSearch*). The *IncrementModel* and the *MovementModel* are the objects that will obtain the *Increment*s or the *Movement*s necessary to modify the *Solution* (under construction or not).

The *IncrementModel* and the *MovementModel* may change at runtime, reflecting the use of a dynamic neighborhood in the *LocalSearch*, or having a *BuildStrategy* that uses several kinds of *Increment*s to construct a *Solution*. The variation of the *IncrementModel* is controlled inside the *BuildStrategy* and the variation of the *MovementModel* is controlled by the *LocalSearch*. This control is performed using information made available by the *Client* and accessible to these objects. Figure *2* illustrates this scenario.

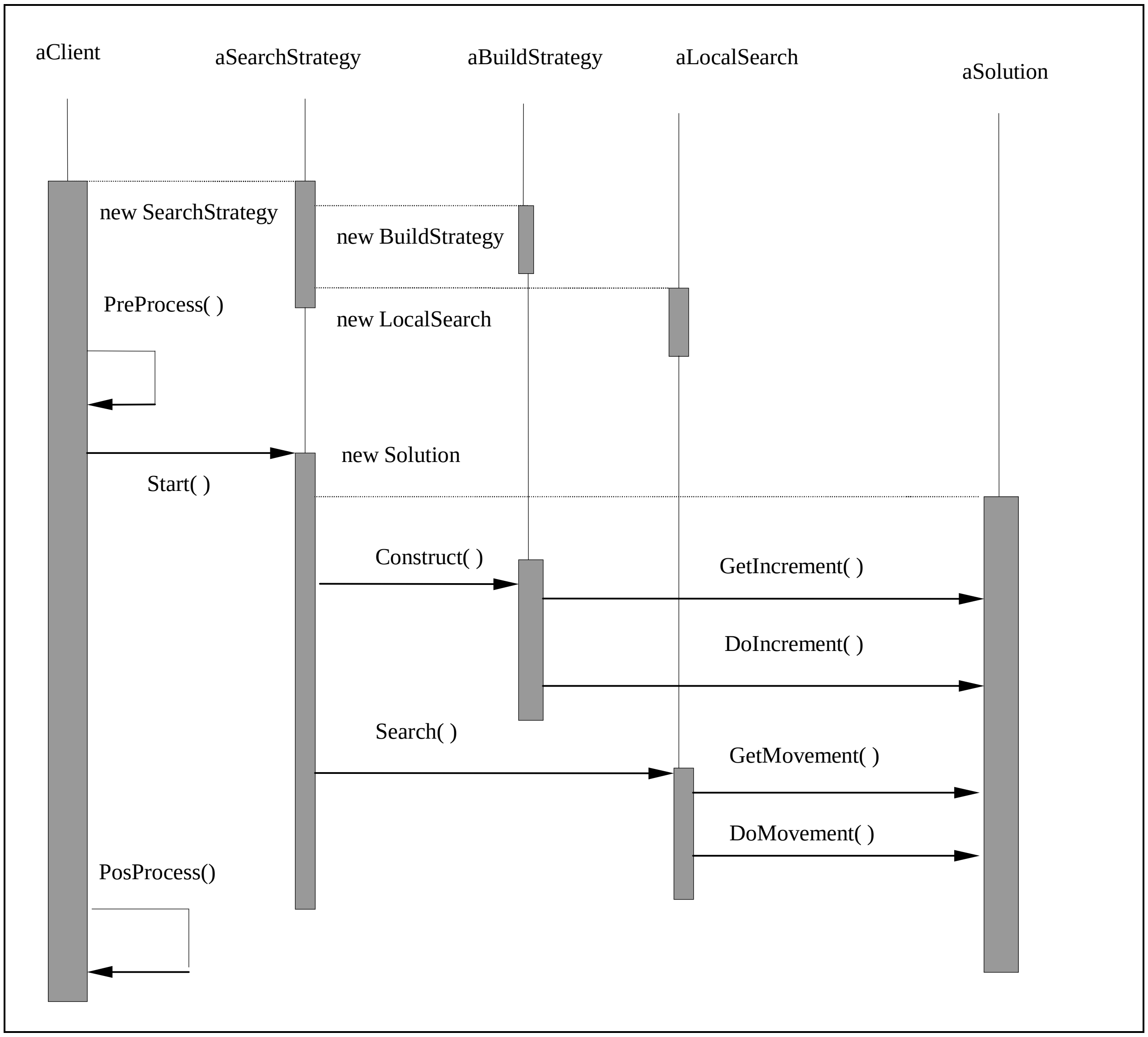

Figure 2. Collaborations in the Searcher framework

3. SEARCHER DESIGN MODEL IN UML-F

Although the pattern-based description gives an intuition of the framework design structure, there are several problems related to it[1]:

- Variation point identification: Variation points (or hot-spots) (Pree, 1995) are the points in the framework structure that are designed to be replaceable. Frameworks instances are created through the adaptation of the variation points. In the OMT diagrams shown in Figures 1 and 2 there is no indication of what are the variation points and how they should be adapted. The textual description is informal and might not be clear enough.

[1] It is important to remember that in the initial version of the Searcher project (Andreatta et. al., 1998) UML-F was not yet available.

- Complex design model: the design diagram presented in Figure 1 is very tangled and hard to be understood.
- Interrelated variation points: there are variation point interdependencies that are not represented in the standard class and sequence diagrams. For instance, whenever new build strategies are defined new increment models also need to be. The same holds for the search strategies and the movement models.

Figure 3 represents the Searcher design structure through an extended class diagram, in which the *{variable, static}* tags are attached to method names. The *{variable}* tag is an UML-F construct useful to identify methods that model variation points and need to be implemented during framework instantiation. The *{static}* tag indicates that the instantiation needs to be done statically, meaning that it requires the recompilation of the system for the changes take effect. Another possibility would be the use of the tag *{dynamic}*, meaning that the instantiation would be possible during runtime. However, the use of *{dynamic}* in the design model would require an interpreted implementation language that allows dynamic class loading (such as Java and Smalltalk), since generally *{dynamic}* variation points cannot be directly implemented in complied languages such as C++.

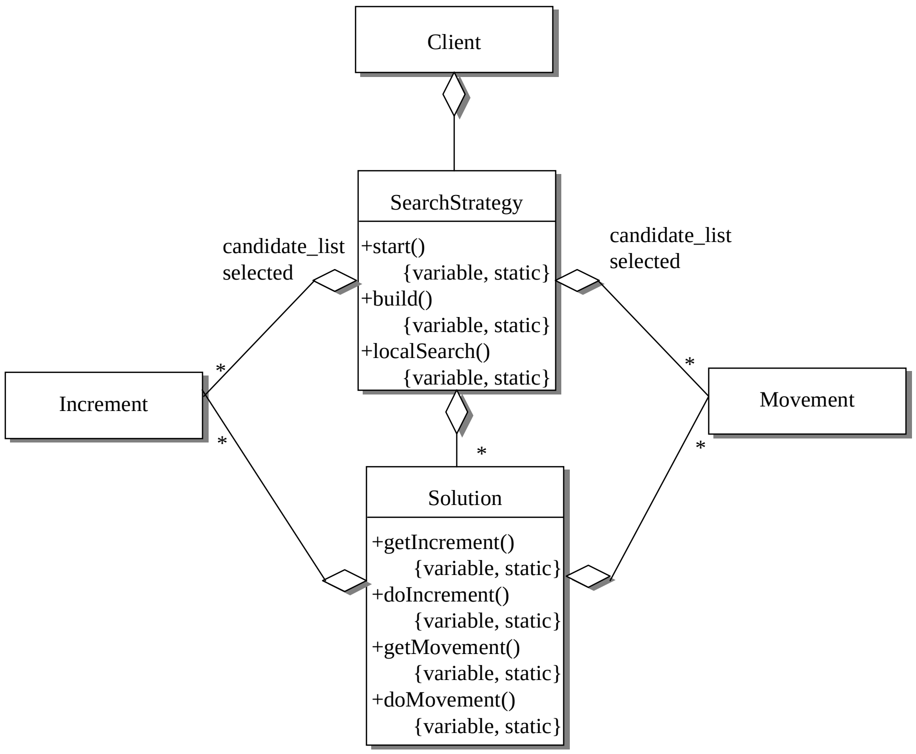
Client
SearchStrategy
+start() {variable, static}
+build() {variable, static}
+localSearch() {variable, static}
candidate_list
selected
candidate_list
selected
Increment
Movement
*
*
*
*
*
Solution
+getIncrement() {variable, static}
+doIncrement() {variable, static}
+getMovement() {variable, static}
+doMovement() {variable, static}

Figure 3. UML extended class diagram for the Searcher framework

Figure 4 illustrates the instantiation diagram for the Searcher framework. Instantiation diagrams provide a representation of the instantiation process through the use of UML activity diagrams. Activity diagrams are used to represent workflow procedures in standard UML (Rumbaugh et. al., 1998; OMG, 1999). In instantiation diagrams each action state represents one variation point. The transitions indicate the variation points interdependencies, describing the way they should be instantiated.

Several facts related to the framework instantiation may be derived from this diagram. The variation points *build(), getIncrement(),* and *doIncrement()* are interrelated. For each adaptation of *build(),* one or more adaptations of *getIncrement()* and *doIncrement()* have to be defined. The same holds for *search(), getMovement(),* and *doMovement().* Finally, the diagram specifies that the *start()* variation point always have to be intantiated with exactly one definition of the *start()* method per framework instance.

Extended class diagrams (Figure 3) and instantiation diagrams (Figure 4) complement each other. Together, they completely specify variation points and how the framework should be instantiated.

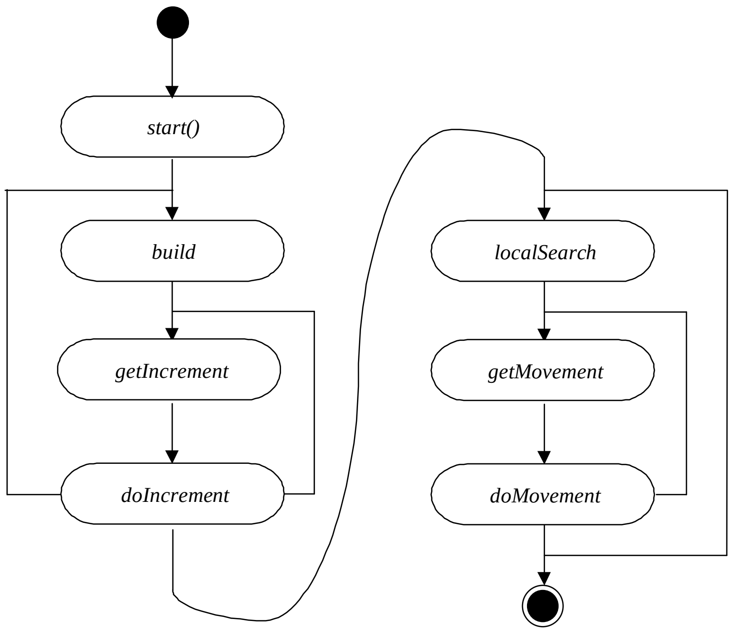

Figure 4. Searcher instantiation diagram

Note that, now, the variation points are explicitly documented, the design is more clear (four classes have been eliminated) and the variation point interdependencies are formally documented in the instantiation diagram. In fact, Figures 1 and 3 complement each other: the first provides a more concrete description of the framework implementation, while the second better models the variation points and their instantiation requirements. Figure 3 is more abstract than Figure 1, which can be seen as an implementation refinement based on design patterns. The stereotype «realize», defined in UML 1.3 (OMG, 1999), specifies the relationship between a specification model and a model that implements it. In this case we can say that Figure 3 is a specification model and Figure 1 is a implementation model that realizes it through the use of design patterns.

The implementation of variation points is one of the most critical parts in framework development. Several techniques can be used, such as design patterns (Buschmann et. al., 1996; Gamma et. al., 1995), meta-level programming (Kiczales et. al., 1991), and contracts (Helm et. al., 1990; Holland, 1993). However, the selection of the most appropriate technique for each of the framework's variation points may be a very difficult task. If the variation points and their properties are not explicitly represented, this task can become even harder.

Once the variation points have been documented in UML-F, the application of the most adequate technique to model each variation point may be automated through design transformations. These transformations are realization transformations that specify how each variation point in a UML-F diagram, modeled by a *{variable, static}* tag for example, can be implemented in actual programming languages such as Java and C++.

Figure 5 shows a logic program that applies the Strategy design pattern (Gamma et. el., 1995) to all variation points defined in the design diagram shown in Figure 3. The result of this transformation is the diagram presented in Figure 1. The transformation illustrated in Figure 5 is part of the framework development tools proposed in (Fontoura, 1999), which assist the development and instantation of frameworks using UML-F. The tool stores UML-F diagrams as graphs in a knowledge base, and provides several logic programs for transforming these graphs. In this example, it searches for all methods marked as *{variable, static}* tags transforming them into the Strategy design pattern model. Figure 6 illustrates this transformation visually for the *localSearch()* variation point.

```
applyStrategy(Project, NewProject) :-
          [...]
          forall(variationMethod(Project, Class, Method, dynamic),
          strategy(Project, NewProject, Class, Method)),
          [...]

strategy(Project, NewProject, Class, Method) :-
          concat(Method, 'Strategy', NewClass),
          createClass(NewProject, NewClass, dynamic),
          createMethod(NewProject, NewClass, Method, public, none, abstract),
          createAggregation(NewProject, Class, NewClass, strategy),
          assert(implementation(Project, [...], strategy)).
          [...]
```

Searches for variation points in a design

Uses strategy pattern to model them

Figure 5. Design-implementation transformation: using Strategy to model variation points

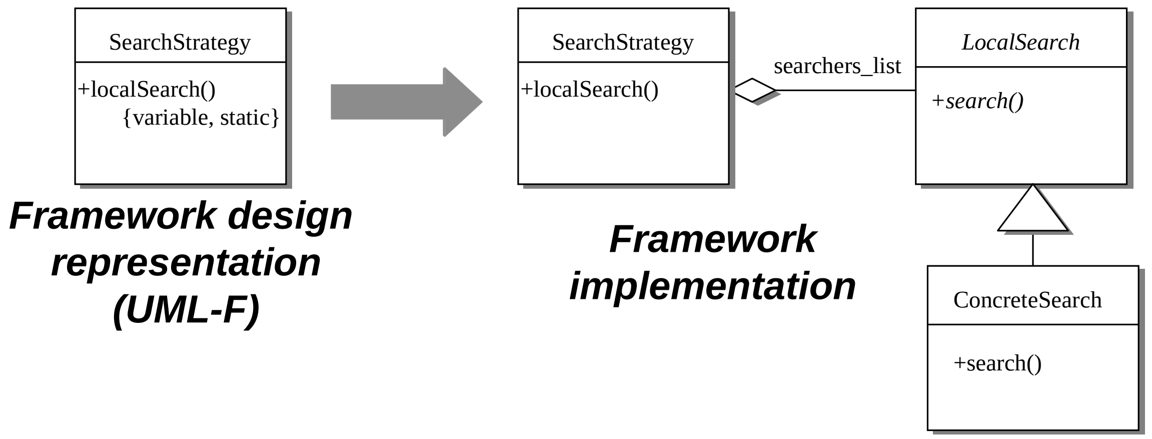

Figure 6. Visual transformation of the design

We think that framework builders should not focus on design patterns, but instead on the domain variability and extensibility requirements. Once this knowledge is captured, it should be expressed in appropriate design notation, such as UML-F. Design patterns and other implementation techniques should be considered only in the implementation step of the process, once the design is completely validated by the domain experts. The above example shows that tools can support these ideas by concentrating at the design level and systematizing the mapping from design to implementation.

Moreover, the instantiation diagram can be used as a formal representation for the instantiation process. It completely specifies the tasks that should be performed by the framework users while adapting the framework. Since all the variation points that need to be adapted are marked in the design, the UML-F diagrams can be processed by tools that use this description for guiding the users during the adaptation. A prototype of such a tool is described in (Fontoura, 1999).

4. RELATED WORK

UML uses collaboration diagrams to model design patterns and provides a way of instantiating pattern descriptions through the «binding» stereotype. Frameworks are represented in UML as packages that assemble several patterns (Rumbaugh et. al., 1998; OMG, 1999). Collaboration diagrams are useful for documenting framework adaptation. However, they only partially address this problem since they do not provide, for instance, a way of describing interdependencies among variation points. We think that collaboration diagrams can be usefully complemented with UML-F diagrams to further assist framework users in during the instantiation process.

UML 1.3 also provides the concept of subsystem. A subsystem offers an interface and can contain specification and realization (or implementation) models. Therefore, frameworks can be represented in UML 1.3 as subsystems. In this case the UML-F diagrams are used to describe the specification models of the subsystem, while standard UML are used to describe its realization models.

Catalysis proposes a methodological approach for developing frameworks based on standard UML diagrams (D'Souza and Wills, 1997). Frameworks are represented in Catalysis as collaborations, and the adaptation process is specified through the use of substitutions. However, there is no direct support in Catalysis for assisting the implementation of variation points and for guiding the instantiation process.

Cookbooks (Krasner and Pope, 1988) provide a textual description of the purpose of the framework, describing its major components and providing examples of its use. However, cookbooks do not have a formal structure, and different cookbooks may focus on different aspects of the framework with different levels of detail. The descriptions are quite informal, and this lack of formality may lead to misconceptions by the user. Although the patterns described in (Johnson, 1992) are based on an Alexandrian narrative, they respect a pattern form and can be seen as a cookbook with more structure.

Riehle and Gross (1998) proposes an extension of the OOram methodology (Reenskaug et. al., 1996) to facilitate framework design and documentation. They propose the use of role diagrams to explicit document the interaction of the framework with its clients. Although this approach produces good results for pattern documentation, it does not provide an explicit representation for variation points and it does not model the instantiation process.

Some work in the systematic application of patterns to implement framework variation points can be found in (Pree, 1995; Schmid, 1997).

5. CONCLUSIONS AND FUTURE WORK

This paper has shown that an adequate notation for frameworks can be useful in automating the implementation and instantiation steps of the software development process. The case study has shown how framework instantiation can be largely improved by the use of UML-F notation. It has also shown that representing frameworks at a higher level of abstraction than programming languages can be very useful to assist with the understanding of a design.

In this paper we have briefly described how tool support for UML-F can assist framework development and adaptation. A more detailed description on that topic and more case studies of the use of the UML-F to describe and implement real world frameworks can be found in (Fontoura, 1999).

A new version of this tool that provides cooperative work capabilities and graphical representation is now being developed in Java. This tool will be used to experiment with different objet-oriented models and evaluate the impact of these models in framework design. More concretely, the tool will provide a uniform way of validating how roles (Riehle and Gross, 1998; Reenskaug et. al., 1996) and ADVs (Cowan and Lucena, 1995) can enhance framework design and how they can be used together with UML-F.

REFERENCES

Andreatta, A., Carvalho, S., and Ribeiro, C., An Object-Oriented Framework for Local Search Heuristics, 26th Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA'98), *IEEE Press*, 33-45 (1998).

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons (1996).

Cowan, D. and Lucena, C., Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse, *IEEE Transactions on Software Engineering*, 21(3), 229-243 (1995).

D'Souza, D. Sane, A., and Birchenough, A., First-class Extensibility for UML – Packaging of Profiles, Stereotypes, Patterns, UML'99, *LNCS 1723*, Springer-Verlag, 265-277 (1999).

D'Souza, D. and Wills, A., *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison Wesley (1997).

Fontoura, M., A Systematic Approach for Framework Development, Ph.D. Thesis, Computer Science Department, Pontifical Catholic University of Rio de Janeiro (1999).

Fontoura, M., Pree, W., and Rumpe, B., UML-F: A Modeling Language for Object-Oriented Frameworks, ECOOP'2000, *LNCS 1850*, Sringer-Verlag, 63-82 (2000).

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

Helm, R., Holland, I., and Gangopadhyay, D., Contracts: Specifying Behavioral Composition in Object-Oriented Systems, OOPSLA/ECOOP'90, Norman Meyrowitz (ed.), *ACM Press*, 169-180 (1990).

Holland, I., The Design and Representation of Object-Oriented Components, Ph.D. Dissertation, Computer Science Department, Northeastern University (1993).

Johnson, R., Documenting Frameworks Using Patterns, OOPSLA'92, *ACM Press*, 63-76 (1992).

Kiczales, G., des Rivieres, J., and Bobrow, D., *The Art of Meta-object Protocol*, MIT Press (1991).

Krasner, G., and 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).

OMG, OMG Unified Modeling Language Specification v.1.3 (1999).

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

Reenskaug, T., Wold, P., and Lehne, O., *Working with objects*, Manning (1996).

Riehle, D. and Gross, T., Role Model Based Framework Design and Integration, OOPSLA'98, *ACM Press*, 117-133 (1998).

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Clifs (1994).

Rumbaugh, J., Jacobson, I., and Booch, G., *The Unified Modeling Language Reference Manual*, Addison-Wesley (1998).

Schmid, H., Systematic Framework Design by Generalization, *Communications of the ACM*, 40(10), 48-51 (1997).

Author biographies:

Marcus Fontoura is a post-doctoral researcher at the Department of Computer Science, Princeton University, Princeton, USA.

Carlos J. Lucena is a full professor at the Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil.

Alexandre Andreatta is an assistant professor at the Department of Applied Computer Science, University of Rio de Janeiro, Rio de Janeiro, Brazil.

Sergio E. Carvalho is an associate professor at the Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil.

Celso C. Ribeiro is a full professor at the Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil.