Using Domain Specific Languages to Instantiate Object-Oriented Frameworks

Marcus Fontoura§, Christiano Braga*, Leonardo Moura*, and Carlos Lucena*

♣ 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
{cbraga, moura, lucena}@inf.puc-rio.br

ABSTRACT

Prior research has shown that high levels of software reuse can be achieved through the use of object-oriented frameworks. An object-oriented framework captures the common aspects of a family of applications, and thus, allows the designers and implementers to reuse this experience at the design and code levels. Despite of being a powerful design solution, frameworks are not always easy to use. This paper describes a technique that uses domain specific languages (DSL) to describe the framework variation points and therefore syntactically assure the creation of valid framework instances. This approach allows framework users to develop applications without worrying about the framework implementation and remaining focused on the problem domain. In addition, the use of DSLs allow for better error handling, when compared to the standard approach of adapting frameworks by directly adding subclasses. The DSL programs are translated to the framework instantiation code using a transformational system. The approach is illustrated through two real-world frameworks.

KEY WORDS: object-oriented frameworks, domain-specific languages, variation points, framework instantiation, transformational systems.

1. INTRODUCTION

Object-oriented frameworks and product line architectures have become popular in the software industry during the 1990s. Numerous frameworks have been developed in industry and academia for various domains, including graphical user interfaces (e.g. Java's Swing and other Java standard libraries, Microsoft's MFC), graph-based editors (HotDraw, Stingray's Objective Views), business applications (IBM's San Francisco), network servers (Java's Jeeves), just to mention a few. When combined with components, frameworks provide the most promising current technology supporting large-scale reuse [17].

A *framework* is a collection of several fully or partially implemented components with largely predefined cooperation patterns between them. A framework implements a software architecture for a family of applications with similar characteristics [26], which are derived by specialization through application-specific code. Hence, some of the framework components are designed to be replaceable. We call these components the variation points or hot-spots [27] of the framework. Moreover, an application based on such a framework not only reuses the frameworks source code, but also its architecture design. This amounts to a standardization of the application structure and allows a significant reduction of the size and complexity of the source code that has to be written by developers who instantiate a framework [11].

The bad news is that framework usability can be a problem. The most common way to instantiate a framework is to inherit from abstract classes defined in the framework hierarchy and to write the required instantiation code. However, it is not always easy to identify which code is needed to instantiate a framework and where it should be written since class hierarchies can be very complex. Application developers have to rely on extra documentation to be able to create framework instances properly, since it is very unlikely that they will be able to browse the framework classes and write the required instantiation code if no adequate documentation is provided.

Another problem is that framework instantiation can be far more complex than simply plugging components into variation points. Variation points might have interdependencies [1, 9], might be optional [9, 35], frameworks may provide several ways of adding the same functionality [35], and so on. These restrictions can make the instantiation process even more difficult.

This paper presents a new process for framework instantiation, which consists in programming the missing information for the variation points in domain specific languages (DSLs) [2, 4, 7, 18, 24]. The DSL programs are then transformed to complete the code for the variation points, generating the desired application. The use of the DSLs allows the designer to focus on the problem domain, while the framework code (and implementation details) remains encapsulated. The DSLs can be used to assure that all instantiation restrictions are preserved and also allow for better error handling (see Section 3). The experiments presented in this paper show that this approach leads to a more effective instantiation process.

The rest of this paper is organized as follows. Section 2 provides a brief introduction to DSLs. Section 3 describes how DSLs may be used to assist framework instantiation. Section 4 describes two case studies that illustrate the proposed approach. Section 5 describes some related work. Finally, Section 6 presents our conclusions and future research directions.

2. DOMAIN SPECIFIC LANGUAGES

A domain specific language is used to formally specify software designs [18]. It is a formal language that is expressive over the abstractions of an application domain. Common examples of DSLs are:

- Tex and Latex, text formatting languages;
- YACC, a parser generator;
- Mathematica, an extensible language for mathematical modeling;
- Schema description and query languages for databases.

An advantage of using a DSL is that the domain expert can express domain specific concepts directly, rather than encoding them in a verbose way, via a wide-spectrum language such as C++. This allows the domain expert to formalize the specification of a software solution immediately instead of communicating the specification informally to a software specialist who may be less familiar with the intended application.

A DSL allows a non-verbose, straight and simple specification of domain concepts. The semantics of a DSL is done via a mapping from the DSL being specified to another language, which has (ideally) a well-defined semantics. In the framework instantiation process proposed in this paper, DSL programs are mapped to the language used to implement the framework (framework language). This approach leads to an operational semantics of a DSL specification in terms of wide-spectrum languages.

3. THE PROPOSED APPROACH

One of the major problems of directly using the framework implementation language (Java, for example) to instantiate the framework is that it does not know what are the variation points and how they should be instantiated. This has some undesirable consequences: (i) there is no indication of where the application developer should write the instantiation code or what code should be written (ii) the implementation language compiler cannot verify the instantiation restrictions, being unable to provide appropriate error messages.

A DSL that “knows” the framework design structure can solve these problems since its syntax can indicate exactly what code should be written. The DSL compiler can realize semantic checks and generate more appropriate error messages. The IBM San Francisco, for example, is a framework for distributed business applications (http://www.software.ibm.com/ad/sanfrancisco/) that is written in Java and its standard instantiation process is trough Java programs that implement the missing variation point information. Some of its variation points have been provided for defining database queries. Since queries are not first-class citizens in Java, there is no way to provide good error report for the San Francisco application developers if their queries are not well defined, for example. If a query language is provided, support for the instantiation of those variation points is vastly improved. Figure 1 illustrates this example.

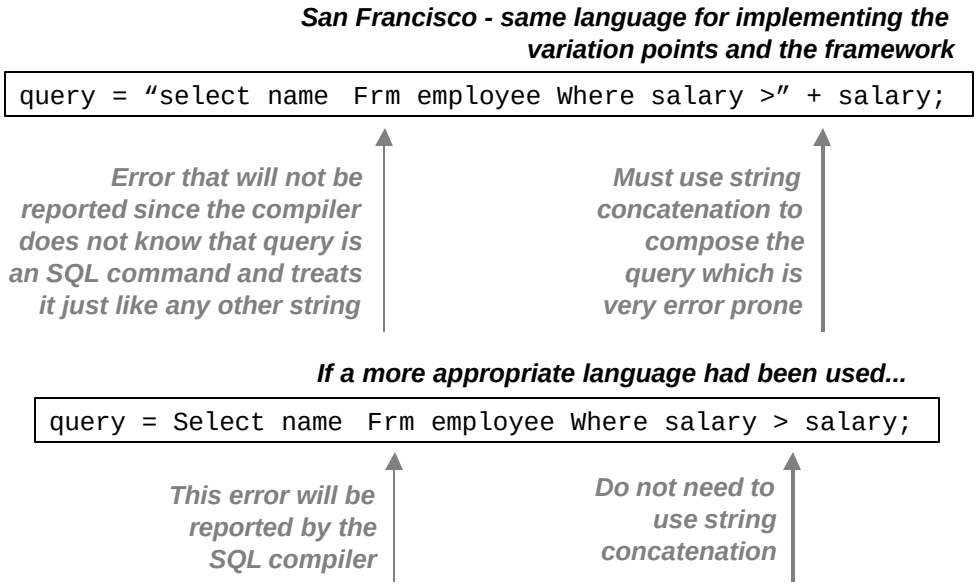

Figure 1. Using DSL to specify queries better than in Java

It may be the case that each variation point is better instantiated by a different DSL. However, several times variation points are better implemented in the framework language (which is the language used to implement the framework itself). In the case that DSLs are used, they have to be transformed (or compiled) to the framework language. Before actually generating the application, it may be convenient to check if any framework instantiation restriction has been violated. One example is to verify if all the required variation points have been implemented. These verifications may be performed during the transformation (or compilation) process, as shown in Figure 2. This structure is similar to the one proposed in aspect-oriented programming (AOP) [19], which is briefly described in section 5.

One of the main advantages of this approach is that the framework code remains encapsulated and the instantiation restrictions may be verified. Note that instantiating variation points with the framework language may require some knowledge about the framework code. This situation may be avoided by the definition of appropriate DSLs. The definition of the most adequate DSL for each variation point is a creative task that cannot be completely automated. Some tools that partially automate this process are described in [8].

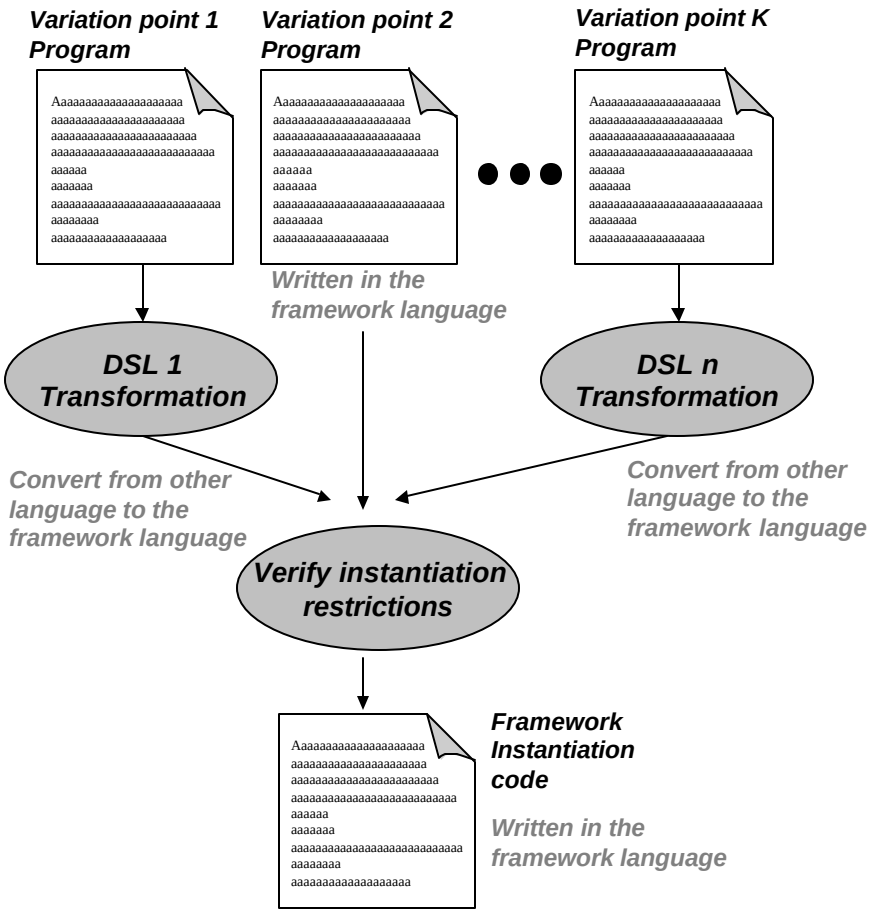

Figure 2. Combining several DSLs to instantiate frameworks

4. CASE STUDIES

This section presents two case studies to illustrate the proposed approach. The first is a framework for local search heuristics [1, 10] while the second is an e-commerce framework [29]. Both case studies are simplifications of real-world frameworks.

4.1 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 [1].

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 [1] 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.

This section describes the framework using a variation of the pattern form proposed in [14] and OMT diagrams [31], as it was first documented by its authors [1, 10].

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 3 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.

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).

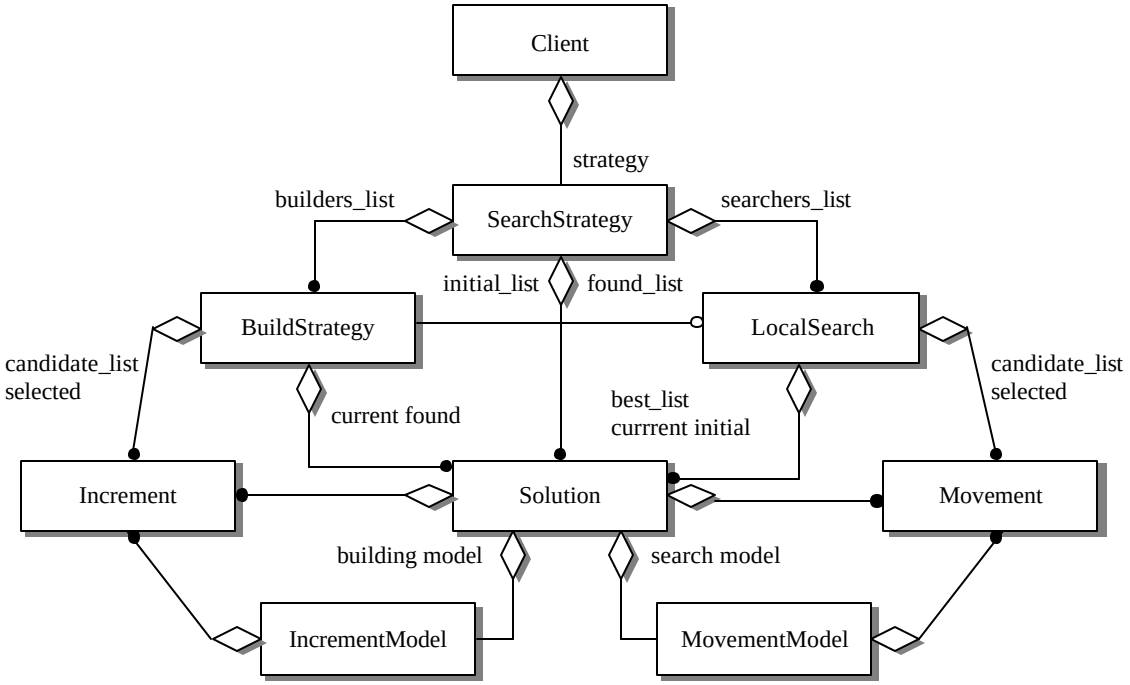

Figure 3. Searcher class diagram

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 LocalSearch controls the variation of the MovementModel. This control is performed using information made available by the *Client* and accessible to these objects.

Figure 4 illustrates this scenario.

Although the pattern-based description gives an intuition of the framework design structure, there are several problems related to it that may encumber the instantiation process:

- Variation point identification: there is no indication in the diagrams of the framework variation points and how they need to be instantiated. The textual description is informal and might not be clear enough.
- Complex design model: the design diagram presented is very tangled and hard to be understood.
- Interrelated variation points: there are variation point interdependencies that are not represented in the diagrams. Whenever new build strategies are defined new increment models also need to be. The same holds for the search strategies and the movement models. The state diagram illustrated in Figure 5 models these dependencies, where each state is a variation point and the transitions indicate how the instantiation process should be performed. This diagram can be seen as a formal cookbook [21] for instantiating Searcher.

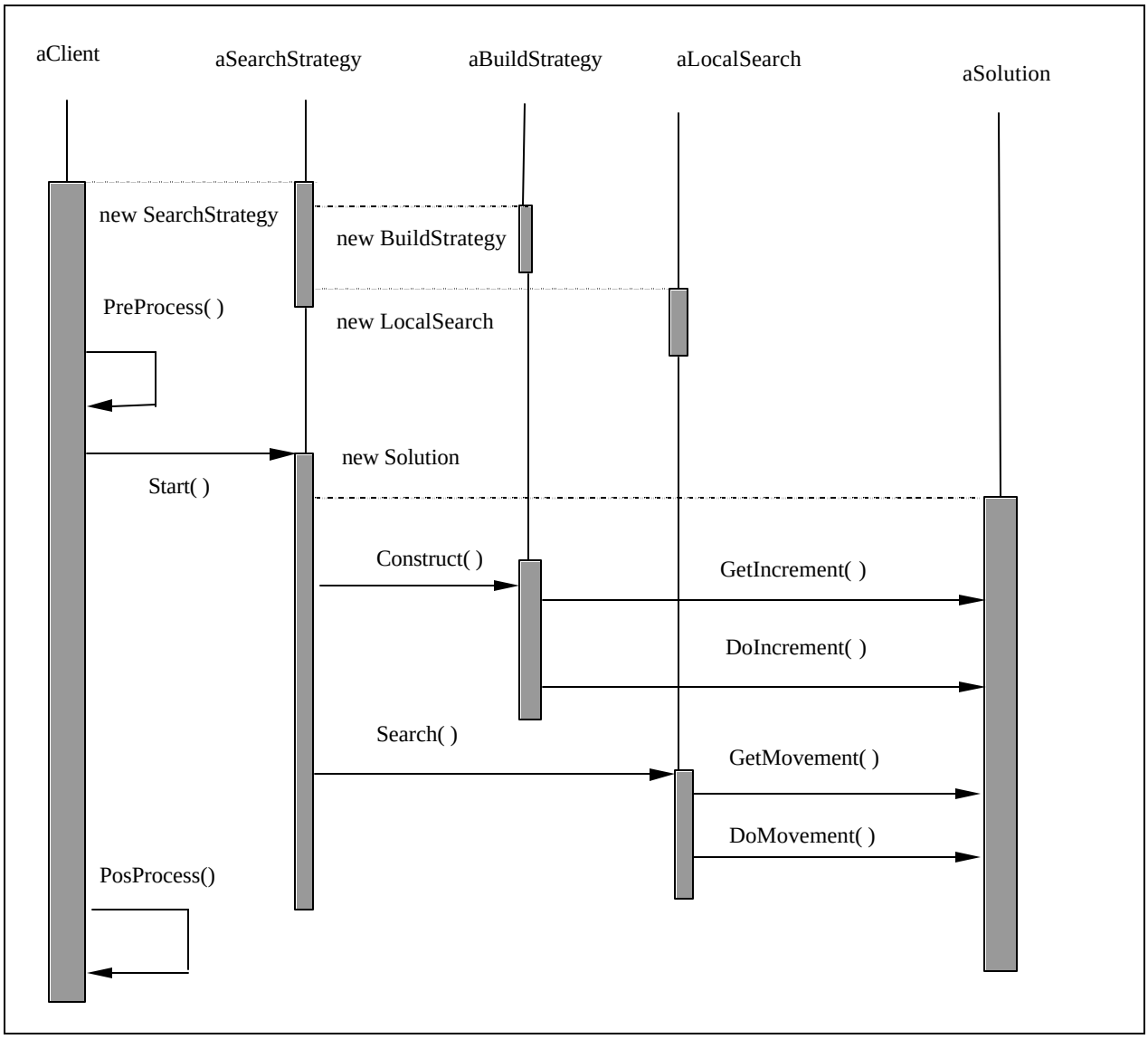

Figure 4. Collaborations in the Searcher framework

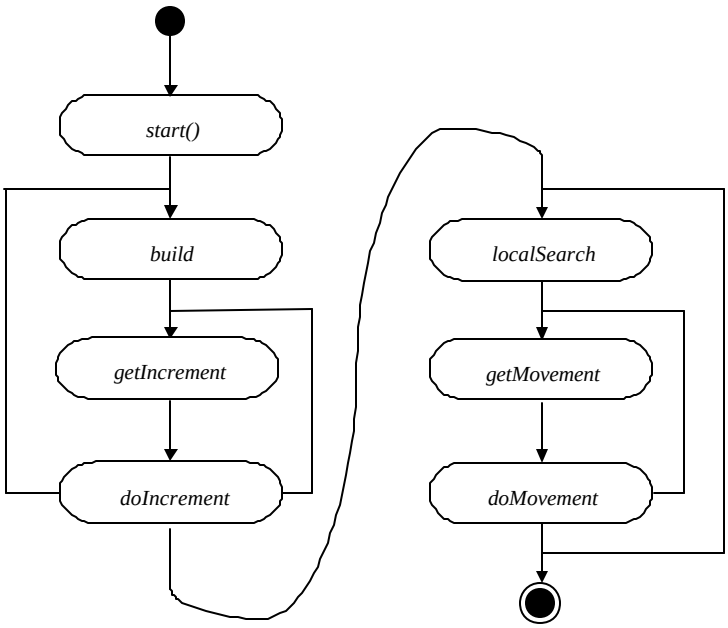

Figure 5. Searcher variation point interdependencies

The *start()* variation point is responsible for invoking a certain combination of build and search strategies, therefore it can be specified in a DSL by just providing the list of build and search strategies that are going to be used. These lists are translated to complete the *start()* variation point code: they are exactly the *builders_list* and *searchers_list* in the C++ code illustrated in Figure 6.

```
List<Solution> SearchStrategy::Start(Client * owner ){
   BuildStrategy builder;
   LocalSearch searcher;
   List<Solution> initial_list;
   ...
   for( ; ; ){
     builder = builders_list.Next( );
     initial_list.Append( builder.Construct( owner ) );
     if( builders_list.IsEmpty( ) ) break;
   }
   for( ; ; ){
     searcher = searchers_list.Next( );
     for( ; ; ){
       found_list.Append(searcher.Search(initial_list.Next()));
     if( initial_list.IsEmpty(  )) break;
     }
    if( searchers_list.IsEmpty( )) break;
   }
  ...
}
```

Figure 6. Start C++ code, completed by the builders and searchers lists

The other variation points may be structured into two groups *{build, getIncremet, DoIncrement}* and *{search, getMovement, doMoviment}.* Since they have similar structure, and consequently similar design solutions, lets analyze just the second group. The search variation point is implemented with the Template Method design pattern [14], as illustrated by the C++ code shown in Figure 7. The code shows that the instantiation of this variation point would require the definition of the *StopCriteria()* and *Selection()* methods. This instantiation is exemplified by the two instances: *InterativeImprovement* and *TabuSearch*.

```
class LocalSearch{
   public:
      ...
      List<Solution> Search( Solution initial );
   protected:
      Solution current;
      ...
      virtual Boolean StopCriteria( );
      virtual Movement Selection( List<Movement> movement_list );
}
List<Solution> LocalSearch::Search( Solution initial){
      Movement selected, obtained ;
      List<Movement> candidate_list;
      ...
      for ( ; ; ){
         if ( StopCriteria( )) break;
         ...
         for ( ; ; ){
            if ( current.NbhAlreadyScanned( )) break;
            obtained = current.GetMovement( );
            candidate_list.TryInsertion(obtained);
         }
         selected = Selection(candidate_list);
         current.DoMovement(selected);
         ...
      }
}
class IterativeImprovement: public LocalSearch{
   protected:
   Boolean StopCriteria( );
   Movement Selection(...);
   ...
};

class TabuSearch: public LocalSearch{
   protected:
   Boolean StopCriteria( );
   Movement Selection(...);
   ...
};
```

Figure 7. Implementation for Search based on template method

Another important aspect of *search* is that it requires the definition of the *getMoviment* and *doMoviment* variation points (represented in the C++ code by *current.GetMoviment()* and *current.DoMoviment()*). Therefore, instantiating the group *{search, getMovement, doMoviment}* requires the definition of four C++ methods. Moreover the methods have to be defined in subclasses of *LocalSearch* and *MovimentModel*, which have no interaction with the rest of the framework.

The solution for assisting the instantiation of this group is the definition of the four C++ methods used for each search strategy in a single place. This solution is illustrated in Figure 8, which exemplifies the DSL approach for instantiating the entire framework. The transformation process is responsible for generating code for the *start()* method and for checking the instantiation restrictions. All the other methods are specified in C++ and checked only by the C++ compiler. Note that with this solution its is possible to perform domain specific checks, such as "A search method is defined without its corresponding *doMoviment()*", "No search strategy is defined", "At least one build strategy specified in start is not implemented", and so on.

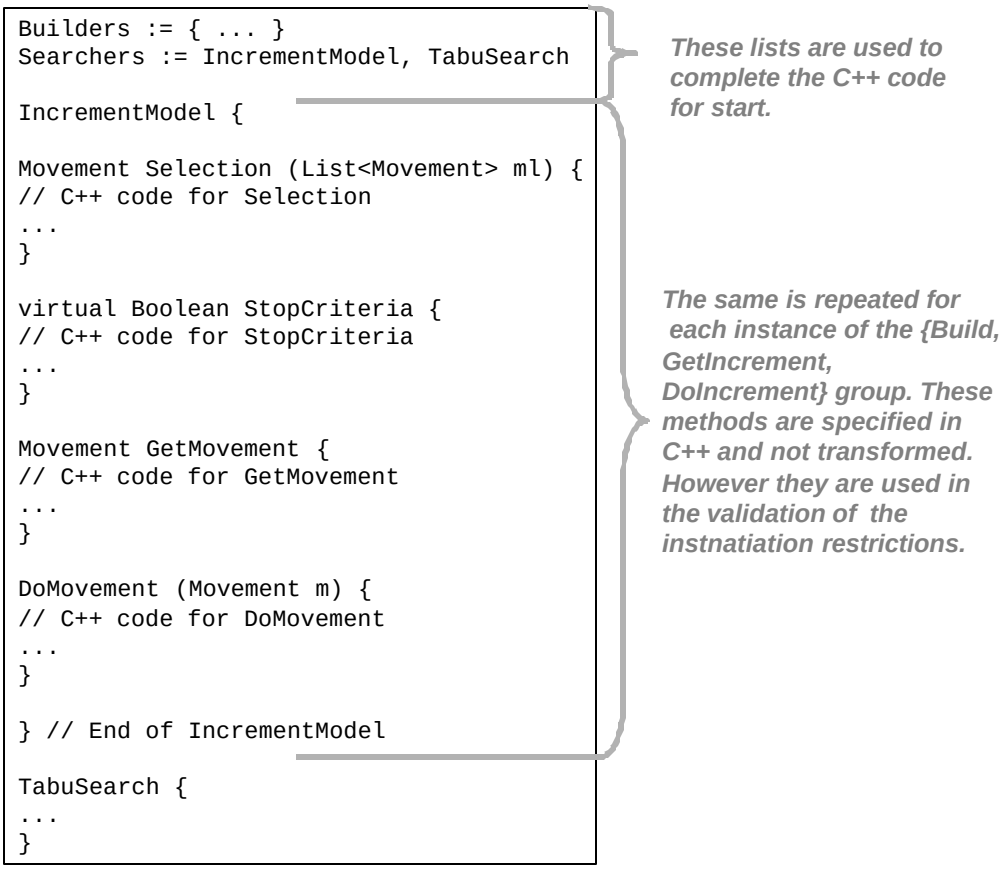

Figure 8. DSL-based approach for the Searcher instantiation

The main advantage of this approach is that frameworks users do not need anymore to understand the complex design structure for Searcher. They just need to fill the language template shown in Figure 8 and launch the transformation process. The analysis we have made so far show that the use of this approach can reduce at least by half the time to understand and instantiate Searcher.

4.2 THE VIRTUAL-MARKEPLACES (V-MARKET) FRAMEWORK

V-Market is greatly inspired on Media Lab's Kasbah [5] system, and concentrates mainly on the ability to create applications based on virtual marketplaces, where buying and selling agents interact. V-Market is a powerful experimentation and research tool, which allows for the fast development of new robust marketplace applications in a fairly simple way [29]. The most important variation point in the V-Market is the support for multiple types of products.

One of the main problems faced with the current implementation of Kasbah [5] is that it is completely tied to the two types of goods that it now supports (books and CDs). To add a new type of product it is necessary to make a major change to the system's structure. The current implementation of the buying and selling agents and the persistence scheme are extremely tied to the specific description of each item.

V-Market addresses these issues by allowing for new types of products to be easily added to the virtual marketplace. For this to be possible, item definitions should be generic enough to support not only commodity type of products such as books and CDs but also intangible ones, such as knowledge about a specific subject, skills, or services. In addition, a standard item description/structure must be developed so that agents do not need to be redesigned for every new item added to the marketplace.

To achieve this goal, each item in the framework must be able to store and compare its attributes. Figure 9 illustrates the design for this variation point, which is based on meta-programming [20]: a meta-object protocol (MOP) was developed to allow the definition of new items. Each *Item* is defined as a list of *MetaItem* objects.

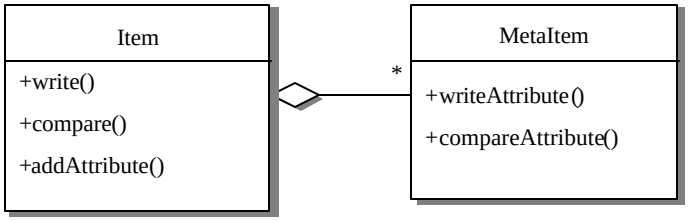

Figure 9. Item MOP

To assist the instantiation of this variation point we have used XML. A supporting tool parses an XML description of the new instances and generates the HTML files that will interface with the end user and creates the new items using the MOP methods. Figure 10 illustrates the DTD used in the instantiation process.

```
<!ELEMENT ITEM (NAME,DESCRIPTION, ATTRIBUTE+)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT DESCRIPTION (#PCDATA)>
<!ELEMENT VALUE (#PCDATA)>
<!ELEMENT LABLE (#PCDATA)>
<!ELEMENT DIRECTIONS (#PCDATA)>
<!ELEMENT ATTRIBUTE (NAME, LABEL, DIRECTIONS?, DESCRIPTION, PRESETS*) >
        <!ATTLIST ATTRIBUTE ATYPE (text|number) "text">
        <!ATTLIST ATTRIBUTE COMPARISON
        (equal|similar|no|numericalBigger|numericalSmaller) #REQUIRED>
        <!ATTLIST ATTRIBUTE INPUT_TYPE
        text|textarea|combobox|radio|checkbox)#REQUIRED>
        <!ATTLIST ATTRIBUTE BROWSEBLE  (yes|no)#REQUIRED>
        <!ATTLIST ATTRIBUTE REQUIRED  (yes|no)#REQUIRED>
        <!ATTLIST ATTRIBUTE SIZE CDATA "45">
<!ELEMENT PRESETS ((VALUE,LABLE)+|(VALUE)+)>
```

Figure 10. Item DTD

The XML tool generates high-level error messages, such as “Attribute type not defined” and “Item already defined”, which could never be provided by the Java compiler, which is the framework language. We estimate that the use of this tool reduces the instantiation time for this variation point at least by a factor of three [29]. Researchers at the Software Engineering Lab (LES) at PUC-Rio are now developing similar tools for the other V-Market variation points.

5. RELATED WORK

“Aspects” are cross-cutting[1] non-functional constraints on a system, such as error handling and performance optimization. Current programming languages fail to provide good support for specifying “aspects,” and consequently the code that implements them is typically very tangled and spread throughout the entire system. Aspect-oriented programming (AOP) [19] is a technique proposed to address this problem. An application that is based on the AOP paradigm has the following parts: (i.a) a component language, used to program the system components, (i.b) one or more aspect languages, used to program the aspects, (ii) an aspect “weaver”, which is responsible for combing the component and the aspect languages, (iii.a) a component program that implements the system functionality using the component language, and (iii.b) one or more aspect programs that implement the aspects using the aspect languages.

[1] If there are two concepts that are better represented in different programming languages (such as code optimization and the logic of the system itself) they are said to be cross-cutting concepts [19].

It is valid to think that the framework kernel may be implemented in the component language, and each variation point in a different aspect language, as shown in [8]. If this approach is taken, the "weaver" is also responsible for verifying the instantiation restrictions.

The hook tool [12, 13] is in fact a process-based tool, which enacts the hook definitions, which describe how the variation points should be instantiated. This approach is also interesting since it assures that all variation points are properly instantiated during the process execution. A similar approach based on UML [25] descriptions for the instantiation process is proposed in [8].

The wizards in existing integrated development environments [3, 23] can be seen as a primitive kind of framework instantiation tool. In these environments the user interacts with a sequence of dialog boxes in order to generate code. The main drawback is that all generators are hard-wired in these systems, and it is not possible to define new abstractions.

HP Labs has a successful reuse program based on domain-specific kits [15]. Domain-specific kits [16] are designed to help the development of families of related applications and achieve software reuse. A domain-specific kit is the combination of an application framework, a set of reusable components, and glue languages, which may be DSLs and/or wide-spectrum languages such as Java or Smalltalk. The glue languages can be used to connect the components to variation points or to generate the missing code. The kits may also be composed of other tools (e.g. builders and wizards), documentation, and examples. However, there is no indication in their work of how the kits are designed and what are the builders/DSLs being used to support framework instantiation.

Several other authors propose the use of DSL to instantiate frameworks [4, 30] but none of them describe how DSLs should be composed and how to verify framework instantiation restrictions.

A work for enhancing the utilization of software libraries is described in [28, 34]. The motivation for their work is that end users often do not make effective use of libraries, because most of them lack the expertise to properly identify and compose the routines appropriate to their application. They argue that in domains with mature subroutine libraries, one can greatly improve the productivity and quality of software engineering by automating the effective use of those libraries. The DSL generator approach can be seen as an enhancement of this approach for frameworks. A main difference has to do with the technique used to generate the application. In [28, 34] the code is generated based on a deductive approach [22] using constructive theorem proving. In our technique the code is generated using DSL compilers or transformational systems with generative component concepts [6, 32, 33].

6. CONCLUSIONS AND FUTURE WORK

This paper presents an approach to integrate frameworks with domain specific languages (DSL). We argue that DSLs allows the domain expert to formalize the specification of a software solution immediately without worrying about implementation decisions and the framework complexity. The code for the variation points is specified in DSLs that are transformed (or compiled) to generate the framework instantiation code. During the transformation the framework instantiation restrictions may be verified. The case studies have shown that the proposed approach may enhance very much the instantiation process.

It is important to note that DSLs can be transformed into other DSLs, thus creating a domain network, in a way similar to that described in [24], providing an easy implementation path for new DSLs. An approach for the derivation of the framework instantiation restrictions based on UML specifications is shown in [8], as well as tool support for the transformations. We are now working on a more elaborated version of the supporting environment, based on UML case tools and specific transformational systems [8].

ACKNOWLEDGMENTS

The authors would like to thank Alexandre Andreatta, Celso Carneiro Ribeiro, and Sergio Carvalho (in memoriam), for assisting us with the Searcher case study, Marcelo Sant'Anna for his comments on early versions of this paper, and Pedro Ripper for developing the V-Market case study.

REFERENCES

1. A. Andreatta, S. Carvalho, and C. Ribeiro, “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.

2. J. Bell, “Software design for reliability and reuse: A proof-of-concept demonstration”, In TRI-Ada ’94 Proceedings, *ACM Press*, 396-404, November 1994.

3. Borland Inc., *Delphi User’s Guide*, 1995.

4. J. Bosch and Y. Dittrich, “Domain-Specific Languages for a Changing World”, (http://bilbo.ide.hk-r.se: 8080/~bosch/ articles.html).

5. A. Chavez and P. Maes, “Kasbah: An Agent Marketplace for Buying and Selling Goods”, Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'96), London, UK, April 1996.

6. J. Cordy and I. Carmichael, “The TXL Programming Language Syntax and Informal Semantics”, Technical Report, Queen’s University at Kingston, Canada, 1993. (http://www.qucis.queensu.ca/STLab/TXL).

7. E. Dijkstra, “The humble programmer”, Communications of the ACM, 15(10), October 1972.

8. M. Fontoura, “A Systematic Approach for Framework Development”, Ph.D. Thesis, Computer Science Department, PUC-Rio, 1999 (http://www.cs.princeton.edu/~fontoura).

9. M. Fontoura, L. Moura, S. Crespo, and C. Lucena, “ALADIN: An Architecture for Learningware Applications Design and Instantiation”, Technical Report MCC34/98, Computer Science Department, PUC-Rio, 1998.

10. M. Fontoura, C. Lucena, A. Andreatta, S. Carvalho, and C. Ribeiro, “Using UML-F to Enhance Framework Development: a Case Study in the Local Search Heuristics Domain” Journal of Systems and Software, to appear 2001.

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

12. G. Froehlich, H. Hoover, L. Liu, and P. Sorenson, “Hooking into Object-Oriented Application Frameworks”, ICSE’97, *IEEE Press*, 491-501, 1997.

13. G. Froehlich, H. Hoover, L. Liu, and P. Sorenson, “Requirements for a Hooks Tool”, (http://www.cs.ualberta.ca/~softeng/papers/papers.htm).

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

15. M. Griss, “Systematic Software Reuse: Objects and Frameworks are Not Enough”, Object Magazine, February 1995.

16. M. Griss and K. Wentzel, “Hybrid Domain-Specific Kits”, Journal of Systems and Software, February, 1995

17. D. Hamu and M. Fayad, “Achieving Bottom-Line Improvements with Enterprise Frameworks”, *Communications of ACM*, 41(8), 110-113, 1998.

18. P. Hudak, “Building Domain-Specific Embedded Languages”, ACM Computing Surveys, 28(4es), 196-es, 1996.

19. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, ECOOP’96, *LNCS 1241*, 220-242, 1997.

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

21. G. Krasnes and S. Pope, “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”, *Journal of Object-Oriented Programming*, 1(3), 26-49, 1988.

22. Z. Manna and R. Waldinger, “Fundamentals of Deductive Program Synthesis”, *IEEE Transactions on Software Engineering*, 18(8), 674-704, 1992.

23. Microsoft Inc., *Microsoft Visual C++ 4.0 User’s Guide*, 1995.

24. J. Neighbors, “The Draco Approach to Constructing Software from Reusable Components”, *IEEE Transactions on Software Engineering*, 10(5), September 1984

25. OMG, “OMG Unified Modeling Language Specification V.1.2”, 1998 (http://www.rational.com/uml).

26. D. Parnas, P. Clements, and D. Weiss, “The Modular Structure of Complex Systems”, *IEEE Transactions on Software Engineering*, SE-11, 259-266, 1985.

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

28. T. Pressburger and M. Lowry, “Automatic Domain-Oriented Software Design using Formal Methods”, Software Systems in Engineering, Energy-Sources Technology Conference and Exhibition, 33-42, 1995.

29. P. Ripper, “V-Market: A Framework for Agent Mediated E-Commerce Systems based on Virtual Marketplaces”, M.Sc. Dissertation, Computer Science Department, PUC-Rio, 1999.

30. D. Roberts and R. Johnson, “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.

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

32. M. Sant’Anna, “Transformational Circuits”, Ph.D. Dissertation, Computer Science Department, PUC-Rio, 1999 (in Portuguese).

33. M. Sant'anna, J. Leite, A. do Prado, “Draco-PUC: A Workbench For Developing Transformation-Based Software Generators”, ICSE'98, *IEEE Press*, vol. 2, 135-139, 1998.

34. M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwook, “Deductive Composition of Astronomical Software from Subroutine Libraries”, 12th Conference on Automated Deduction, 1994.

35. J. Vlissides, “Generalized Graphical Object Editing”, Ph.D. Dissertation, Department of Electrical Engineering, Stanford University, 1990.