Describing Framework Static Structure: promoting interfaces with UML annotations

Sérgio Lopes, Adriano Tavares, João Monteiro, Carlos Silva
Department of Industrial Electronics
University of Minho, Campus de Azurém
4800-058 Guimarães, PORTUGAL
Email: <sergio.lopes,adriano.tavares,joao.monteiro,carlos.silva>@dei.uminho.pt

***Abstract*—Frameworks are an important form of reuse that can help to significantly decrease the time and cost of application development. Although widely known, there are still some difficulties associated with framework reuse, which are critical to its success. In this paper, we focus on the issues regarding the framework reuse process, and more specifically, on the framework architecture description.**

This paper discusses our position on the subject. It enables a component-oriented approach to framework reuse by emphasizing the description of black-box variation-points, and introducing *call-points*. We define the UML-FD profile for UML 2.0, which extends UML to support these and other concepts, dedicated to describe the static structure of application frameworks.

I. Introduction

Reuse has been a major concern for software engineers, in their quest for easier application development with reduced time-to-market and cost. Frameworks are an important form of reuse that can help to get closer to this long-time pursued goal. However, several problems associated with frameworks have been identified [2], starting from the framework development and ending at its maintenance. This work is concerned with the difficulties inherent to framework reuse from the perspective of the application developer.

Object-oriented frameworks [1] are widely known, but building applications by reusing them poses problems that software developers have to struggle with. Frameworks capture a specific domain's commonalities and variabilities, by implementing common elements and providing an architecture that localizes variability at *variation-points*. In contrast with passive forms of reusable software (typically, procedural libraries or traditional class libraries), frameworks are active and exhibit predefined behaviour, which imposes some control flow among its components. Consequently, they are often complex and hard to understand, what can make reusing them a difficult and time-consuming assignment [1].

These problems must be alleviated in order to make framework reuse an alternative way of building applications that is decisively attractive. In fact, it is not always guaranteed to be advantageous comparatively to application development by "reinventing the wheel". The typical framework slow learning curve is one major issue because it results in a delayed productivity payoff, which can arrive unrewardingly late or even be unacceptable. Therefore, diminishing these problems is a decisive factor for the success of framework-based reuse approach. It has been consensually recognized the need to effectively communicate frameworks and provide appropriate tool support, in order to minimize the effort and time required to build applications. The present paper focuses on difficulties in communicating the framework to the re-user, and our proposal to lessen them.

Although communicating the architecture is a framework developer responsibility, the notation chosen is decisive to the following phases of the reuse process to be carried out by application developers. It is widely accepted that visual notations have crucial advantages over textual languages in the communication of software. Being a *de-facto* standard object-oriented design notation in industry, UML [21] is a beneficial choice for describing frameworks, but it is a general scope and extensible language, not specifically tuned for this purpose. It has been previously demonstrated [4] the need to explicitly represent frameworks variation-points, not supported by UML, in order provide effective framework description. Actually, applying a subset of UML to object-oriented frameworks reuse has been addressed before, with a few dedicated extensions being proposed [4], [8], [9]. However, a few limitations still endure and we investigate how to tackle them, in order to provide a more complete support for the framework reuse needs. We propose the UML-FD profile that integrates a different perspective, namely, promoting a component-oriented approach to application frameworks.

This paper describes on going research, and starts by discussing some issues regarding framework description in following section. In section III, we introduce our perspective and discuss the foundation of our proposal. In section IV, the requirements for object-oriented notations supporting framework reuse are outlined. The UML-FD profile, defining the proposed UML extensions, is described in section V. In section VI, the related work is reviewed and our contribution is explained. Finally, we present the concluding remarks.

II. About Framework Reuse

A framework can be reused in many different ways that require different kinds and amounts of information, which may be constrained by business interests. The support that tools can offer is also affected, but this is not a matter for this paper.

We discriminate two fundamental forms of object-oriented framework reuse: unanticipated reuse and anticipated reuse. The differences of them are given below, in a discussion that describes their relationships with framework documentation aspects.

A. Anticipated Reuse vs. Unanticipated Reuse

We make the distinction between these two forms of reuse because the information needs, the activities and the results of each one are quite different.

Anticipated reuse takes place when the particular needs of a re-user are fulfilled by the functionalities of the framework. In more concrete terms, the framework provides enough *variation-points* (also named *hot-spots* [3] or *hooks* [17]) with enough flexibility, to cope with a re-user objectives. Or, the other way around, the re-user goals can be achieved by a subset of all the possible variation-points' adaptations. The framework adaptation is realized by providing application specific components for variation-points that observe the respective constraints. With this kind of activity, typically, the re-user does not have to worry about possible erroneous interactions between the framework components.

Unanticipated reuse happens when the re-user wants to add some functionality that is not provided by the framework components, or only to make a slight change to some feature. Usually, these kind of goals cannot be achieved solely by adapting the framework's variation-points. Most probably, it will be necessary to make adaptations outside the set of pre-defined variation-points. By doing it, the re-user can more easily introduce erroneous interactions between application specific components and the framework. Furthermore, these flaws can be difficult to correct, as we see next.

B. Description Information and Business Rules

Telling apart the two forms of reuse above is important because they have a strong impact on both the kind and quantity of necessary information about the framework, which in turn is a subject of business concerns.

Anticipated reuse is easier to document because the anticipated variability is localized at variation-points. Therefore, it is sufficient to provide detailed design documentation only about them. In particular, describing the purpose of each variation-point, how to adapt it, and their semantic restrictions which guarantee that the adaptation will work correctly. Tools can also be built to provide specific assistance for filling the variation-points. In contrast, unanticipated reuse can occur at almost any part of the framework. Hence, besides the variation-point description, it requires detailed design (and possibly implementation) information about the complete framework. Notice this is generic software documentation, because it is not possible to provide specific reuse information. The same applies to tools, which cannot provide any special development support for unanticipated reuse.

Communicating a precise and deep understanding of the framework to the re-user is essential to enable the assessment of viability that is necessary to achieve specific goals, and the development of adaptations that do not violate the framework architecture. Naturally, the description should be independent of implementation details not important for design, which might limit the framework generality, or lead to complications caused by framework evolution (see [19] for this kind of problems). These issues apply to both kinds of reuse, but are much harder to manage when support for unanticipated reuse is intended. Unanticipated reuse requires complete information about the framework, in order to enable the re-user to develop unpredicted adaptations, which correctly interact with the framework, and/or change parts of it while maintaining behaviour consistent with untouched parts.

Furthermore, the problem with detailed architecture description, necessary for supporting unanticipated reuse, is that it may collide with business interests. There are a few free open-source frameworks but, on the other hand, there are commercial frameworks provided by vendors. Development of a framework is a long and costly process that requires high expertise in the target domain. Thus, revealing the architecture details is not usually considered good business because it may give advantages to competitors in the same market, and framework vendors may be suppliers of dedicated support tools. For these reasons, if a framework is not open and there is no detailed documentation about it, it may prove to be very difficult to achieve a successful unanticipated adaptation.

C. Description Techniques and Reuse Possibilities

The spectrum of approaches for framework documentation can be classified according to two categories: informal prescriptive techniques, and formal descriptive techniques.

Prescriptive techniques [14], [15], [16], [17] describe how to use the framework, normally using natural language, or other informal means of documentation. They provide valuable guidance but only to the limited adaptation possibilities described. It is not possible to predict all the ways of adapting a framework, in fact, not even is feasible to describe a large number of them. Therefore, these techniques are more suited to support anticipated reuse. They are also oriented towards less skilled users, or to enable experienced users a quicker first application build.

Descriptive techniques describe the framework architecture, usually using formal or semi-formal visual languages, like UML [4], [8], [9]. They do not dictate or elicit any particular way of reusing it; instead, they try to communicate the framework architecture to the user. They do not provide significant guidance for adapting a framework; it is up to the re-user to figure out how to adapt it, in order to meet her/his specific requirements. Consequently, these techniques are appropriate to support unanticipated reuse, and are more oriented towards experienced users, who need detailed information more than guidance.

III. Our Approach

Before we get to the proposed solution to communicate frameworks, we explain our point of view about framework reuse, which is the foundation for it. First, the technique chosen to describe frameworks, and then the perspective on reuse technology, are presented.

We follow the same line as the UML techniques cited above, i.e., investigating how to augment the reuse flexibility that descriptive techniques provide, with as much guidance as possible. In agreement with the exposed in the previous section, adopting a descriptive technique to describe a framework in detail supports unanticipated reuse. Augmenting a general descriptive technique with support for explicit variation-point description enables to provide guidance for anticipated reuse. Variation-points are the typical key concept for organizing this kind of documentation for frameworks, but they are not enough, as it will be argued in the next section. No special requirements apply to the description of a framework for unanticipated reuse, it is much like describing any other piece of software. Therefore, the requirements for OO notations for framework description (presented below) reflect only the part of reuse that is anticipated, because it is the one that requires a dedicated approach.

White-box frameworks [5], more than object-oriented, are class-oriented because their adaptation is frequently based on inheritance (sub-classing framework classes), which is a mechanism that describes class hierarchies. We favour a predominantly black-box approach in which the framework is reused by calling its interfaces and providing components that implementing the interfaces it requires. This approach emphasizes use relationships instead of inheritance, and thus is suitable to support the representation of object interactions. In turn, this also facilitates the specification of restrictions on clients.

When using a framework, an application developer is reusing both a design and its implementation. Therefore, we choose not to abstract the variation-point description to the design level, as opposed to [4]. In fact, we consider that the framework should be described with as much precision as possible (without neglecting what was stated in the previous section). Design variations-points defined by inheritance can be refactored into use relationships, with interfaces to be implemented by application specific components [4]. This can be done using the Strategy design-pattern [6], or other suitable design patterns based on separation meta-patterns [3]. This polymorphism and forwarding technique separates the interfaces from implementations, making the design more decoupled and flexible than with inheritance, and it is the base for a black-box approach to reuse.

Moreover, if framework classes provide separate computational and compositional interfaces, it enables a decomposition of the framework instantiation process into two different reuse activities. In the literature, the process of reusing a framework to build an application is usually denominated framework instantiation [1]; we subdivide it in two activities or two phases – framework adaptation, and application instantiation – which we explain next.

The adaptation phase (also designated as 'framework instantiation' in [4]) consists in providing application specific adaptations that define the behaviour of variation-points. The application developer learns the details about the framework architecture from the annotated UML diagrams, and extends it with the application specific components.

Once all components necessary for an application are available, the final application can be defined. This is accomplished in the application instantiation phase, by creating instances of framework and application components, configuring and interconnecting them to form the final executing application. The components provide a compositional interface including methods whose names typically start by 'set', 'add', 'remove', etc, that enable run-time configuration.

A complete discussion of the reasons behind the separation between framework adaptation and application instantiation can be found in [22], which discusses our perspective on tool support for these activities.

To conclude, frameworks designed this way enable the application definition by creating and configuring its run-time units individually, like components. We consider them component-oriented frameworks, because they can be adapted by composing components, although we do not consider any standard component model. We provide support for reusing them, but we also provide specific constructs for white-box reuse that can be useful for describing "gray-box" frameworks.

IV. Framework Description Requirements

Considering the discussion in the previous section, we present below, what we consider to be, the main requirements for describing frameworks in order to facilitate its reuse.

Based on our experience in adapting and implementing frameworks, and on the revision of previously proposed solutions for framework design, we have elicited a requirements list for design languages to describe OO frameworks. First, we present it, and then we discuss each one of the requirements:

1) Domain and purpose of the framework and its specific features;
2) Framework static structure with explicit variation-point identification;
   a. Support for white-box, black-box and client reuse;
   b. Define different types of variation-points with clear semantics;
   c. Variation-point syntax enabling the definition of semantic restrictions on the adaptation;
3) Framework dynamic behaviour with explicit support for variation-points;
   a. Define causal obligations for variation-point adaptation;
   b. Explicit differentiation of variation-points messages in behavioural compositions;
4) Guidance for framework adaptation process, with support for optional variation-points;
5) Guidance for the application instantiation process.

The complete framework documentation should include the identification of its target domain, as much as possible defining the boundaries of that domain, and stating which problem the framework solves in that domain. It should also provide a functional view of the features provided by the framework.

The description of framework static structure identifies the components that compose its design and their relationships. It

gives a static view of the objects' collaborations. It should explicitly distinguish the variation-points from the framework core, in order to assist the framework user in identifying more easily the parts that need to be provided, or adapted, to create applications. This a form of endowing descriptive techniques with some guidance for the framework adaptation phase.

White-box and black-box variation-points have been consensually recognized as forms of adapting frameworks, and their explicit identification and description has been supported by graphical notations dedicated to framework reuse ([4], [8], [9]). Nevertheless, frameworks do not always rely exclusively in the Hollywood Principle ("don't call us we'll call you", or inverted control mechanism based on Template Method [6]) to communicate with application components. Sometimes they provide services to be called by clients, as has been recognized in [18]. We have developed a framework for measurement systems, inspired by [13], that combines predominant 'inverted' flow of control with pieces of non-inverted control flow. This example experience suggests broadening the framework variety to frameworks that have a neither purely called neither purely calling architecture. The interaction between clients and framework through use relationships may vary from single method invocation to complex protocols that impose obligations on the clients. We share this view with [10], which also emphasizes that use relationships, as a basis of behaviour composition, play an important role in framework integration. Therefore, we introduce the notion of call-points as parts of the framework interface, anticipated for client use, that play a key role in the framework operation. We believe call-points are a concept that reflects an important variety of reuse needs and, for this reason, we widen the explicit identification of framework reuse points to support them.

Variation-points should be classified according to different types, more refined than white-box and black-box, providing additional semantics which are helpful for guiding the framework adaptation. Their semantics should be made as precise as possible, with clear description of the abstract possibilities it opens and abstract restrictions it imposes. Their syntax should support the representation of additional semantic adaptation restrictions that may be useful to specify limits to the set of possible application instantiations.

The description of dynamic behaviour gives a view of the dynamic aspects of the framework design that clarifies the objects' responsibilities, their context dependencies, and how they can be combined. By representing explicitly the run-time collaborations between objects, it reveals the framework architecture. How much of this information is provided depends on the factors considered in the previous section. This information is fundamental to comprehend the framework and, once more, it is vital for opening the door to the flexibility of unanticipated reuse. Furthermore, it also enables the description of causal obligations for variation-points and call-points. These behavioural restrictions should be documented, if they exist, and the corresponding messages in object interactions should be explicitly differentiated from the framework core messages.

The adaptation process should be guided by a description that helps to reduce the complexity of the task, especially for medium and large-scale frameworks. Some variation-points may be optional, and others may require the adaptation of another variation-point. These dependencies should be described in order to provide more guidance and facilitate the job of the application developer.

Finally comes the instantiation process, which should also be guided some how. A framework may be adapted to build an application, or to be integrated into a larger project. These processes should be described, if not in abstract, at least with partial, or complete, concrete examples.

V. Static Structure Description

Although the set of requirements in the previous section cover all aspects of framework documentation, this paper deals only with the description of static structure, i.e. corresponding to requirement 2.

UML 2.0 is a convenient choice for describing frameworks due to its widespread use. It provides structural diagrams, which depict the static features of the model, and behavioural diagrams that describe the dynamic aspects of the model. UML structural diagrams include the class, object, package, component, composite structure, and deployment diagrams. To describe the frameworks' static structure several of these available diagrams can be used for different purposes. In our opinion, two diagrams are rather useful: the class diagram for explicit identification and characterization of variation-points and call-points, and composite structure diagrams as a complement to elucidate its architecture.

We introduce the UML Profile for Framework Description (UML-FD), which extends UML with dedicated concepts supporting a few different variation-points and call-points. Naturally, the proposed annotations address the aforementioned requirements 2-a through 2-b, and therefore we do not discuss them further. The UML-FD profile is defined for UML 2.0, i.e. it augments the current version of UML making use of its improved extensibility mechanism.

Fig. 1 defines the profile abstract syntax and its integration with the UML 2.0 meta-model. All the UML meta-classes extended by UML-FD belong to the `Classes::Kernel` language unit. Tables I to III describe the semantics of each individual extension in a compact tabular form (similar to the presentation of UML 2.0 standard stereotypes). Each table groups variations-points according to white-box variation-points, black-box variation-points, and call-points, providing a clear separation between these different reuse categories.

White-box variation-points are supported because they can be useful to describe white-box frameworks, which is classically the first form that every framework assumes. The *application class* annotation is not a variation-point at all, but instead, it can be used to discriminate framework classes from application specific ones. Both *extensible class* and *non-overridable method* follow the Open-Closed Principle of object-oriented design. *Variable method*s are usually abstract methods of abstract classes. The three white-box variation-points can be can be directly implemented by subclasses,

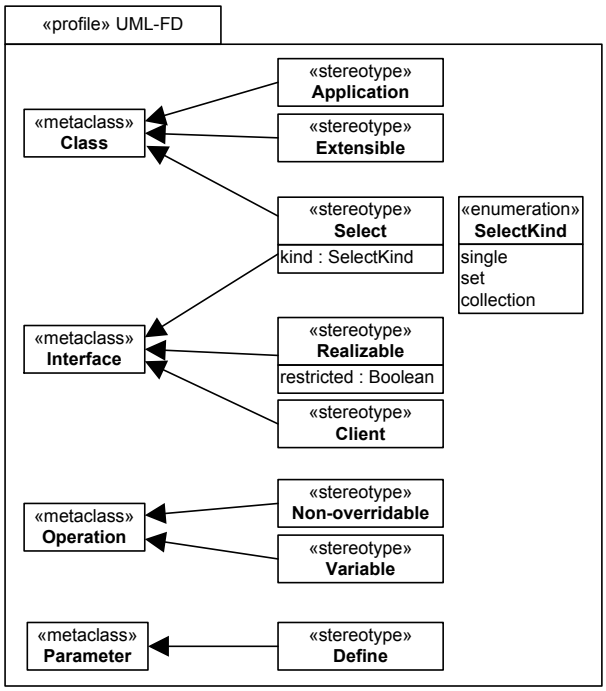

Fig. 1. The UML-FD Profile for UML 2.0.

however that is not recommended. They can also be combined in both concrete and abstract classes.

Black-box variation-points and call-points are the recommended option for reusing a framework. The *realizable interface* is the black-box reuse variation-point with inverted control flow, to be implemented during the framework adaptation phase. All three white-box variation-points can (and should) be converted to a realizable interface, as described in section III. *Select class* and *define parameter* are black-box variation-points which are defined during the application instantiation phase (according to the approach described in section III).

In the next subsections, we discuss in more detail the main contributions of the UML-FD profile to the description of framework static structure.

A. Client Interface

A *client interface* identifies a framework call-point. A call-point is defined by a bidirectional association between one client interface and a framework class that provides one corresponding *service interface*. The association end connected to the framework class identifies (has the name of) a service interface, or *control method*, to be used by the client interface.

The example of Figure 2 illustrates a call-point that is an implementation of the Observer design pattern [6]. The call-point is identified by three elements: the `Observer` interface; the association connecting `Observer` and `Figure` with an association end for `Figure` whose role name is `subject`; and, the part of `Figure` class interface defined by the `Subject` interface. The `Observer` interface is the client interface that must be implemented by client components using the call-point. The `subject` role name of the association end identifies the name of the service interface to be used by the `Observer` client interface. The `Subject` interface is the service interface that defines the `Figure` method(s) to be called by `Observer` client(s).

The client interface construction is a kind of localized role modelling at implementation level, in which client interfaces represent role-types to be integrated by client classes, and service interfaces represent role-types assigned to core framework classes. It enables the modelling of multiple collaborations, through disjunctive groups of semantically related methods (role types), on the same framework interface. Each collaboration is specified by one association that identifies the framework interface methods to be called (service interface) and connects to the respective interface required on clients (client interface). This solution is described only under the perspective of the static structure description: roles, repre-

TABLE I
WHITE-BOX VARIATION POINTS

Applies to	Stereotype	Semantics
Class, Interface	«Application»	An **application class**, or application interface, is part of the application, as opposed to classes which belong to a framework.
Class	«Extensible»	An **extensible class** can have new methods added.
Operation	«Variable»	A **variable method** is a method to be implemented by application classes (implementation variation).
Operation	«Non-overridable»	A **non-overridable method** can be extended but cannot be overridden, *i.e.,* any overriding method must always invoke it.

TABLE II
BLACK-BOX VARIATION POINTS

Applies to	Stereotype	Semantics
Interface	«Realizable»	A **realizable interface** is an abstract type for which application classes can be defined. It has a property named **restricted** that if true forbids sub-typing (classes implementing it, cannot have a different interface).
Class, Interface	«Select»	A **select class** limits the variation-point to the concrete sub-components provided by the framework. It has a property named **kind**, whose value can be **single**, **set**, or **collection**.
Parameter	«Define»	A **define parameter**, is a parameterized variability that defines an important characteristic of the framework (e.g., in opposition to ordinary attributes or parameters related to component interconnection). Constraints on the valid values may be defined as supported by UML (e.g., Enumeration).

TABLE III
CALL-POINTS

Applies to	Stereotype	Semantics
Interface	«client»	A **client interface** is an interface to be implemented by application classes that interact with a framework by calling services of its components (use relationship). Any (*call-back*) methods it has define client constraints, namely static behaviour obligations.
Operation	«control»	A **control method** is a method to be called by clients to externally control some special framework function or trigger some event.

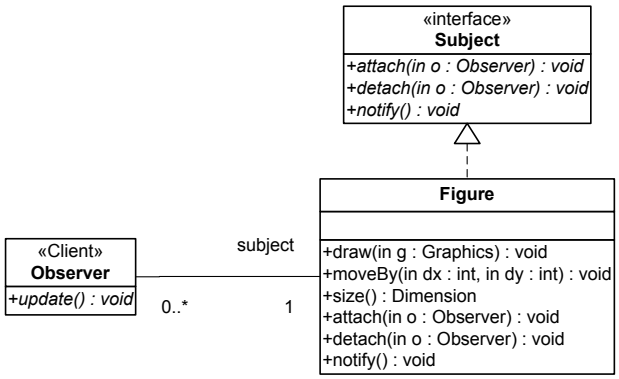

Fig. 2. Example of a Client Interface.

sented by interfaces, describe type information only (or static behaviour). However, client interfaces can be empty, which supports the specification of dynamic behaviour obligations on frameworks clients, independent from structural properties.

The idea behind it is to take advantage of some useful properties of the role modelling technique while avoiding some of its intrinsic verbosity and complexity (see next section properties discussion). Namely, it enables the definition of client restrictions without over constraining client implementation structure. It also provides more structure and semantic information about framework call-points and its relationships with clients. The framework description is kept succinct because client interfaces express role-types which are confined to call-points. This technique avoids the overweight and complexity of the coexistence of reusable role models with respective implementations, by keeping the description at implementation level and within a single paradigm.

B. Control Method

The control method identifies another kind of framework call-point. It is a simpler construction for using framework services that involves a single component method, because it requires no separate service interface for the core framework component. In addition, it is not intended to be used with a client interface, although it might (as defined above). As an example, we have used it to model the trigger function for a real-time embedded framework, shown in Figure 3. The `Sensor` application component invokes the `update()` control method, to stimulate the `Trigger` framework component. Obviously, the framework description does not include the application class, which is included in the figure only for illustrative purposes.

Control methods are applicable more generally to event-driven frameworks that depend on externally fed events. We believe, control methods can also be used to model frameworks that enable easier composition with other frameworks, by providing externally regulated control-flow. They can be used to synchronize the control-flow of such frameworks.

VI. Related Work

The framework description problem has been addressed from informal textual language approaches [7], to formally

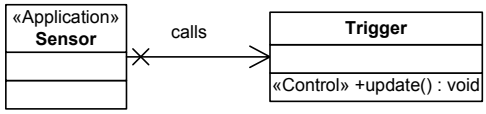

Fig. 3. Example of a Control Method.

defined visual notations that extend UML [8].

Informal textual techniques are usually prescriptive. One first example is the *cookbook* [14] for the Model-View-Controller framework, useful for implementing graphical user interfaces. A similar work is found in [15], where little more structured of a set of Alexandrian-based *patterns* helps to reuse the Hot-Draw framework. Both describe the framework purpose and how to use it. They consist of non-uniform narrated descriptions with minimal structure, and examples solving problems about how to use the framework. This kind of technique was improved by *hooks* [17], which are more uniform, formal and structured adaptation receipts. Hooks define a classification of adaptation methods and kinds of support provided. However, its typology does not provide a clear separation of adaptation activities involved, and they may focus on different framework functionalities with different levels of detail. As discussed in section II.C, prescriptive techniques are focused on the framework intended use and therefore do not offer support for unanticipated reuse.

Although discussing how our approach can be combined with software tool support is outside the scope of this paper, we still look at tool-based solutions, but we concentrate on framework communication and we overlook the facilities for automating framework adaptation. By using software tools, it was possible to improve the cookbook technique to *electronic books*. *Active cookbooks* [20] are a prominent example, which provides interactive receipt descriptions that explain how to use the framework design to solve problems. However, it lacks flexibility because the user has to follow the dictated steps. Evolutions of the electronic book approach are *Smartbooks* [11] and *Specialization Patterns* [12]. Smartbooks are based on a hierarchical interactive hypertext interface through which the desired framework functionalities are chosen. From it, a task plan is generated which guides the adaptation. Specialization patterns are described by a dedicated notation, which lacks tool support. The specialization patterns for a specific framework are embedded in a tool, which handles them providing support for building applications. None of these approaches provides the explicit representation of variation-points within the framework design. Although formalized somehow (to enable tool processing), it is still the framework designer who prescribes its adaptation options.

A few works have been devoted to descriptive visual approaches for documenting frameworks. UML has been the obvious target, being extended with concepts dedicated to framework documentation [4], [8], [9]. These works have similarities – all provide UML extensions to identify variation-points – and parts that are complementary: [4] focus more on variation-points identification and characterization, while [8] provides stronger support for expressing framework syntax and semantics, and [9] introduces selection of black-box com-

ponents and parameterization. The role modelling technique [10] is a complementary technique that tackles object relationships, which are fundamental for framework integration and composition. The requirements on clients calling framework services are explicitly represented but, on the other hand, this approach disregards the explicit identification of variation-points. frameworks may require different instantiation mechanisms. Catalysis [23] also applies UML to support reuse. However, it defines model frameworks as collaborations of abstract types, which are reused through parameter substitution. It does not address the reuse of (code) frameworks, and consequently it does not provide dedicated annotations for explicit representation of its variation-points. Catalysis defines a software development method based on the concepts of model frameworks and components.

Our research also explores UML as visual descriptive techniques for describing frameworks. It builds on previous work, but we provide a wider and more complete coverage of the different reuse needs. While keeping the support for white-box variation-points, a clear and precise definition of black-box variation-points is provided. We introduce UML extensions for explicit expression of use relationships with constraints on clients, to facilitate the reuse and composition of called frameworks [18] and black-box [5] frameworks. We do that by introducing call-points, which borrow inspiration from concepts of the cited role modelling technique. By putting a special emphasis on use relationships, or object relationships, we enable a black-box approach to framework reuse.

VII. Conclusion

Software engineering has pursued for decades the ambition of increased reuse and software quality. Frameworks are an important alternative, which offers high reuse potential, but still have a few problems to be tackled. Addressing these difficulties, namely by employing graphical notations and providing appropriate tool support, it is critical to its success as an option for application development, and for the reuse goal in general.

Some important factors that influence the support that is provided for framework reuse were discussed. An explanation of our perspective on framework reuse was given. A requirements list for object-oriented design notations providing specific support for framework reuse was elaborated and discussed. The UML-FD profile for UML 2.0 was defined, offering a wider coverage of needs for describing framework static structure description. The role modelling technique was analysed in more detail, because it is the background for part of our work.

We have provided a clear separation of different reuse options: white-box, black-box and call-points. Although the proposed notation supports white-box variations-points, we encourage a component-oriented approach by emphasizing back-box variation-points and call-points. For that purpose, we also define how framework interfaces must be described in order to enable a black-box application development.

We have introduced the concepts of client-interfaces and control methods that expand the spectrum of reuse concepts, by including specific points for calling framework services. These concepts are important in the context of black-box reuse and framework integration or composition.

We believe the presented work contributes to facilitate the communication of frameworks. Hence, it also helps to decrease the difficulties and complexity associated with the framework reuse-based development, making it a more attractive, easy and rewarding alternative to develop applications.

Acknowledgements

The authors would like to thank the anonymous reviewers for their contribution to this work.

References

[1] M. Fayad, and D. Schmidt, “Object-oriented Application Frameworks,” in *Communications of the ACM*, vol. 40, no. 10, ACM Press, Oct. 1997, pp. 32–38.

[2] J. Bosch, P. Molin, M. Mattsson, PO Bengtsson and M. Fayad, “Object-oriented frameworks — problems & experiences,” in *Building Application Frameworks — Object-Oriented Foundations of Framework Design*, M. E. Fayad, D. C. Schmidt and R. E. Johnson, Ed. New York, NY: Wiley & Sons, 1999, pp. 55–82.

[3] W. Pree, “Meta Patterns — a means for capturing the essentials of reusable object-oriented design,” in *Object-Oriented Programming*, ECOOP ’94, Tokoro, Mario & Pareschi, Ed. Remo: Springer-Verlag, 1994, pp. 150–162.

[4] M. Fontoura, W. Pree and B. Rumpe, “UML-F: A Modeling Language for Object-Oriented Frameworks," in *Proc.of the European Conference on Object-Oriented Programming (ECOOP’00)*, LNCS 1850, 2000, pp. 63–84.

[5] R. E. Johnson and B. Foote, “Designing reusable classes,” *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22–35, Jun. 1988.

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

[7] L. Murray, D. Carrington and P. Strooper, “An approach to specifying software frameworks,” in *Proc. of the 27th Conference on Australasian Computer Science*, Dunedin, New Zealand, 2004, pp. 185–192.

[8] N. Bouassida, H. Ben-Abdallah, F. Gargouri and A. Ben Hamadou, “Formalizing the framework design language F-UML,” in *Proc. of the 1st IEEE International Conference on Software Engineering Formal Methods*, 2003, pp. 164–172.

[9] T. Oliveira, P. Alencar and D. Cowan, “Towards a declarative approach to framework instantiation,” in *Proc. of the Workshop on Declarative Metaprogramming to Support Software Development of the 17th IEEE International Conference on Automated Software Engineering*, Edinburgh, Sept. 2002, pp. 5–8.

[10] D. Riehle, and T. Gross, “Role model based framework design and integration,” in *Proc. of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA’98)*, Vancouver, Canada, 1998, pp. 117–133.

[11] A. Ortigosa and M. Campo, “Smartbooks: a step beyond active-cookbooks to aid in framework instantiation,” in *Technology of Object-Oriented Languages and Systems*, 25, IEEE Press, June 1999.

[12] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa and J. Viljamaa, “Annotating reusable software architectures with Specialization Patterns,” in *Proc. of the Working IEEE/IFIP Conference on Software Architecture*, August 2001.

[13] J. Bosch, “Measurement systems framework”, in *Domain-specific Application Frameworks*, M. E. Fayad, D. C. Schmidt and R. E. Johnson, Ed. New York, NY: Wiley & Sons, 2000, pp. 177–205.

[14] G. Krasner and S. Pope, “A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80”, in *Journal of Object-Oriented Programming*, 1(3), 1988.

[15] R. Johnson, “Documenting Frameworks using Patterns,” in *Proceedings of the Conference on Object-Oriented Programming Systems,*

Languages and Applications (OOPSLA'92), Vancouver, Canada, 1992, pp. 63–78.

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

[17] G. Froehlich, H. Hoover, L. Liu and P. Sorenson, "Hooking into object-oriented application frameworks", in *Proceedings of the 1997 International Conference on Software Engineering*, Boston, MA, 1997.

[18] S. Sparks, K. Benner and C. Faris, "Managing object-oriented framework reuse," *IEEE Computer*, pp. 53–61, Sep. 1996.

[19] P. Steyaert, C. Lucas, K. Mens, T. D'Hondt, "Reuse contracts: managing the evolution of reusable assets," in *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages and Applications*, San Jose, CA, October 1996, pp. 268–285.

[20] W. Pree, G. Pomberger, A. Schappert and P. Sommerlad, "Active guidance of framework development," *Sofware — Concepts and Tools*, Springer-Verlang, 1995.

[21] Object Management Group (2005, July 4th). *Unified Modeling Language: Superstructure* (version 2.0) [Online] Available: http://www.uml.org.

[22] S. Lopes, C. Silva, A. Tavares and J. Monteiro, "Application development by reusing object-oriented frameworks," in *Proceedings of the IEEE International Conference EUROCON 2005*, Belgrade – Serbia & Montenegro, November 2005.

[23] D. D'Souza and A. Wills, *Objects, Components, and Frameworks with UML: the Catalysis approach*, Addison-Wesley, 1999.