Fully-Reflective VMs for Ruling Software Adaptation

Guido Chari*, Diego Garbervetsky*, Stefan Marr†
*Departamento de Computación, FCEyN, UBA. ICC-CONICET, Argentina †Johannes Kepler University, Linz, Austria
Email: *{gchari, diegog}@dc.uba.ar, †stefan.marr@jku.at

I. Introduction

It has become common for software systems to require or benefit from dynamic adaptation, *i.e.*, to modify their behavior while they are running. Among the existing approaches to this problem, language-level solutions are appealing for scenarios in which fine-grained adaptation is needed, *i.e.*, when the granularity of the modifications is that of individual objects, or for small applications where an architectural solution based on complex middleware is overkill. However, there is no consensus on which of the existing language-level approaches to adopt. A recent survey on self-adaptive systems asks [9]: Is it possible to adopt a single paradigm providing all required abstractions to implement adaptive systems?

To answer this question, Salvaneschi et al. evaluate contemporary reflective systems (RS),[1] aspect-oriented programming (AOP) and context-oriented programming (COP). Since the authors identified strengths and weaknesses for all the approaches, their conclusions were not definite. We advocate that a suitable solution must include abstractions to directly mold the semantics of the whole system, considering both, the application and the runtime level.

In contrast, paradigms like AOP and COP frequently fall into indirect mechanisms of adaptation. The reasons are two-fold. First they were not conceived as general solutions for unanticipated software adaptation. Second, despite theoretically they may approach adaptations directly, their main implementations and tools (pointcut languages, layers, etc) biased the user to think in terms of intercepting execution points and redirecting their execution flow. On the other hand, most RSs do not reify[2] all elements of a language and its implementation [2]. As a consequence, the adaptations concerning these elements are not expressible, or can be achieved only indirectly.

We believe that most of the approaches designed to handle adaptation at the language level (RS, AOP, COP, and even middlewares) were biased by the lack of adaptation capabilities in mainstream VMs.

Based on the fact that reflection has already been identified as a fundamental technique for software evolution [6], in this paper we argue that a VM exposing its whole structure and behavior to applications can provide a uniform solution for adapting systems at the language level and at run time. A fully-reflective execution environment (FREE) is a particular flavor of a VM exposing its whole structure and behavior to applications [2]. We believe this kind of platforms deserves more attention in the context of software adaptation.

[1] Salvaneschi et al. use the less specific term *metaprogramming*.

[2] To reify: model a concept as a first-class entity.

II. Unanticipated Software Adaptation

A. Direct vs. Indirect Adaptations

Direct adaptation is the redefinition of program and VM semantics restricted to the required operations and scope. For instance, changing the layout (memory representation) of specific objects.

Indirect adaptation is done by wrapping around (intercept) the required semantics and redirect the execution flow. For instance, intercepting the program before a method activation and delegate it to new code.

To clarify the difference, Figure 1 depicts a very high-level and simple example. In real applications, adaptations may depend on a complex combination of operations and individual instances. An indirect adaptation would intercept, potentially, all the operations in the whole application and redirect them to an ad-hoc method. This method, depending on the receiver object, would determine whether the operation is allowed.

From our experience, in indirect adaptations: 1) The interception of operations is usually implemented as an over-approximation of the points in the program that need an adaptation. 2) Maintainability is hard because when the application changes the interception points must be updated accordingly. 3) Debugging gets cumbersome because intercepted methods could be polluted with instrumented code. 4) Composing adaptations may lead to complex conditions at each interception point jeopardizing performance. 5) If an operation is not interceptable the adaption could not be performed. For instance, language primitives might not be interceptable.

B. Reflective Systems

Reflection in programming languages is a mechanism for programs to express computations about themselves, enabling the observation (*introspection*) and/or modification (*intercession*) of their *structure* and *behavior* [10] through a set of APIs. When these APIs let clients modify or extend the semantics of the language, they are called metaobject protocols (MOPs) [5].

Adaptation Approach: RS expose two ways to adapt an application's behavior, both fulfilling the direct adaptation definition: 1) Modify reified objects, *i.e.*, classes. 2) Attach metaobjects to individual objects.

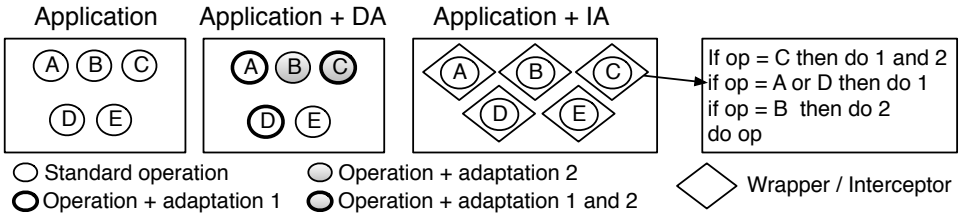

Fig. 1. Five operations and three combination of adaptations approached with both, direct adaptation (DA) and indirect adaptations (IA)

Adaptation Limitations: Even in advanced RSs, reifications [7], [8] cover only a limited subset of the VM entities. As a consequence, they fail to handle directly adaptations demanding changes to low-level entities.

C. AOP

Most AOP [4] implementations typically provide a domain specific language to specify a set of points in the program (*join points*) at which a feature orthogonal to the application logic such as logging, caching, and persistence must be executed.

Adaptation Approach: Pointcut languages facilitates the specification of fine-grained locations where the execution could be redirected to ad-hoc user-defined behaviors.

Adaptation Limitations: Most AOP implementations mostly provide means for adapting applications in an indirect way. For adapting a single operation for a single instance it intercepts all the occurrences and at run time tests whether the actual subject (receiver) of the operation is the required instance. This eventually leads to the problems that indirect adaptations expose (cf. Section II-A).

D. COP

COP is a paradigm specially designed for applications with behavioral variations depending on contextual information [3].

Adaptation Approach: COP languages support the following features for adaptations: a) Means to specify behavioral variations. b) Means to group variations into layers. c) Dynamic activation/deactivation of layers based on context. d) Means to explicitly and dynamically control the scope of layers.

Adaptation Limitations: Adaptations concerning execution semantics or object's structure can not be handled directly. As such, COP is more suitable for dealing with anticipated rather than unanticipated adaptation scenarios.

E. Summary

All the approaches have serious difficulties to support certain adaptation scenarios. Specially, when the adaptations involve VM internals such as object layouts, operational semantics, etc. To handle these scenarios, at best they provide means for indirect adaptations. Our conclusion is that a solution enabling direct semantics adaptations involving both, entities of the application and the VM itself, is still needed.

III. Fully-Reflective Virtual Machines

A FREE[2] is a particular kind of VM providing comprehensive reflective capabilities. Preliminary evidence suggests that this kind of FREE can run efficiently [1]. By design, a FREE enables to express adaptations involving VM entities directly. In addition, a MOP-based FREE enables to describe this semantics in a modular, composable and reusable way, separated from the application's logic. As a consequence, we conjecture that a FREE is a suitable solution for approaching adaptive scenarios at the language level and propose them as a unique solution for software adaptation.

A. Sketching Language-level Approaches in a FREE

Reflective Systems: By definition, a FREE extends, and thus subsumes, RSs because, ideally, a FREE reifies every entity.

Aspect-Oriented Programming: Joinpoints are precise locations of particular operations within the application. Since a FREE can capture any operation and redefine its semantics with language-level methods, it is possible to implement any pointcut language in top of a FREE. On the other hand, one of AOP's most salient features is the decoupling of the crosscutting concerns from the application's logic. MOPs can be designed for supporting the same property by promoting mechanisms for composing metaobjects regarding cross-cutting concerns.

Context-Oriented Programming: The main mechanism to support COP is the redefinition of method lookups and activations so that they take into account the contextual information and the activated layer. By definition, a FREE reifies both concepts. On the other hand, layers just group contextual-dependent behavioral variations. They can still be expressed with any way of grouping methods or even by composing metaobjects.

IV. Conclusions

From our perspective, contemporary reflective systems, aspect-oriented programming, and context-oriented programming present fundamental limitations for handling software adaption in general. The main reason is that, with different degrees of limitations, they do not enable to express direct adaptations for a wide-range of entities. In particular, entities concerning low-level aspects of the system. As a path to follow for the software engineering community, we proposed the incorporation of reflective capabilities to the runtime (VMs) structures. We conjecture that these features are a more suitable foundation for developing flexible software than other language-level approaches.

References

[1] G. Chari, D. Garbervetsky, and S. Marr. Building Efficient and Highly Run-time Adaptable Virtual Machines. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS'16. ACM, 2016. (to appear).

[2] G. Chari, D. Garbervetsky, S. Marr, and S. Ducasse. Towards fully reflective environments. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 240–253, New York, NY, USA, 2015. ACM.

[3] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. pages 220–242. Springer, 1997.

[5] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[6] O. Nierstrasz, M. Denker, T. Gîrba, A. Lienhard, and D. Röthlisberger. Change-enabled software systems. In *Software-Intensive Systems and New Computing Paradigms*, pages 64–79. Springer-Verlag, Berlin, Heidelberg, 2008.

[7] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. ECOOP '02, pages 205–230. Springer, 2002.

[8] D. Röthlisberger, M. Denker, and E. Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Comput. Lang. Syst. Struct.*, 34(2-3):46–65, July 2008.

[9] G. Salvaneschi, C. Ghezzi, and M. Pradella. An analysis of language-level support for self-adaptive software. *TAAS*, 8(2):7, 2013.

[10] B. C. Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 23–35. ACM, 1984.