Component Specialization: Towards Deeper Adaptation

Gustavo Bobeff — Jacques Noyé

École des Mines de Nantes
4, rue Alfred Kastler
44307 Nantes Cedex 3
{Gustavo.Bobeff,Jacques.Noye}@emn.fr

RÉSUMÉ. *De notre point de vue, l'adaptabilité est une caractéristique clef de la notion de composant et devrait être à la base de toute définition de la notion de composant. Cependant, contrairement au modèle de réutilisation en* boîte de cristal *des objets, qui fait l'hypothèse d'un accès complet à l'implémentation de l'objet, cette adaptabilité doit être fortement guidée afin de garder le découplage nécessaire entre le producteur du composant et ses consommateurs, une autre caractéristique essentielle de la notion de composant. Il s'avère que les modèles et infrastructures actuelles de composants ne concilient pas pleinement ces deux caractéristiques et ne fournissent qu'une forme* superficielle *d'adaptation. Tirer tout le bénéfice de la notion de composant demande une forme plus profonde d'adaptation, ce qui requiert de considérer des techniques et outils de spécialisation de programme comme des éléments clefs du génie logiciel à composants.*

ABSTRACT. *To our point of view, adaptability is a key characteristic of components and should be at the heart of any proper component model. However, contrarily to the object* crystal-box *model of reuse, which assumes full access to the object implementation, this adaptability should be strongly guided in order to keep the necessary decoupling between the component producer and its consumers, another key characteristic of components. It turns out that the current component models and infrastructures fall short of conciling these two characteristics and only provide a* superficial *form of adaptation. Taking full advantage of the notion of component requires a deeper form of adaptation, which calls for considering program specialization tools and techniques as key elements of the component-based software engineering toolbox.*

MOTS-CLÉS : *Adaptation, spécialisation de programme, évaluation partielle, découpage, composition en boîte grise, générateur de composant, définition de la notion de composant*

KEYWORDS: *Adaptation, program specialization, partial evaluation, slicing, grey-box composition, component generator, component definition*

1re soumission à *Systèmes à composants adaptables et extensibles*, le 20 juillet 2002.

1. Introduction

The adoption of the component as the unit of design and software construction, instead of an individual class, let us reason in terms of component composition when constructing applications. In this context, an application is conceived as a set of generic components *ready to reuse* connected to each other. But then how do components differ from, let us say, DLLs on the one hand and objects on the other hand ? This is mainly a matter of *adaptability*. *Adaptability* refers to the ability of a piece of software to satisfy requirements dedicated to the context in which it is used.

The DLL model of reuse is a *black-box* model of reuse. The implementation of a DLL, provided as binary code, cannot be changed at all : there is no possibility of adapting a DLL to a specific use. On the other side of the spectrum, the reuse model of standard object-oriented languages is a *crystal-box* model of reuse. It relies on a very good visibility of the implementation of the units of reuses (classes). It is very powerful, as adaptability is almost unlimited, but also very fragile [SNY 86]. Some approaches, like the use of design patterns [GAM 94], make it possible to avoid some of the pitfalls but at the cost of another layer of vocabulary and techniques and therefore additional complexity (and new pitfalls).

A key benefit of introducing a notion of component is to provide a balance between these two extreme visions of reuse. This requires introducing some shade of grey in the initial black-box model in order to provide a very strongly guided form of adaptation. In this paper, we suggest that a general way of doing so is to integrate program specialization techniques to the component development process.

This paper is organized as follows. Sect. 2 gives a basic definition of the notion of component taking adaptation into account. Sect. 3 introduces program specialization and illustrates its application in a component setting. Sect. 4 shows how program specialization can actually be applied using component generators. Sect. 5 concludes the paper.

2. Component Models

Starting from a black-box model, we can say that the *implementation* of a component (implementation refers here to program text, whatever the form of this program text, native code, bytecode, source code ...), provided by a component *producer* is essentially a black box, out of reach of the component *consumer*. However, in order for the component consumer to use a component in a composition an *interface* is provided by the producer. The role of this interface is to guide the composition (rejecting, for instance, incorrect compositions) from two main points of view: a structural point of view (a matter of interconnections) and from a behavioral point of view (a matter of interactions) (see, for instance, [SHA 96, COU 01]). But, on top of this syntactic role, this interface has also a semantic role, being transformed at composition time (at least conceptually) into a *wrapper*, encapsulating the *core* of the component (the result of the transformation of the implementation).

Whereas the implementation is a black box, the interface is a *white box*. It is not a crystal box as it is neither as transparent nor as fragile, but through parameterization facilities, it is possible to change the component and adapt it to the composition.

To keep things simple, we have distinguished here only two phases in the life of a component: its production and its use in a composition. Note that a component can also be *instantiated* (talking about a *component instance* would then be more precise) and finally *executed*. Also, component composition can be incremental and take place either at compile time (the component may then still need to be *deployed*) or at run time. Depending on the details of the model, the interface can be turned into a wrapper at a different time than composition and this transformation could even be incremental.

This general model is actually used in all the industrial component infrastructures such as the Enterprise JavaBeans (EJB) [DEM 00], or COM+ [SES 00]. In particular, it makes it possible to adapt components with respect to predefined technical services (persistence, security ...).

What is however disappointing is that, in this model, there is no adaptation action on the implementation of the component. Along the different phases of the component life many pieces of adaptation information are fed into the interface/wrapper, but this has no effect on the implementation/core. It is however possible to go one step further using program specialization.

3. Program Specialization

In this section, we describe some basic specialization techniques and give a few scenarios showing where and when these techniques can be helpful.

3.1. *Specialization techniques*

Program specialization makes it possible to automatically transform a program fragment into a specialized version, according to an execution context. In this section we describe two well-proven program specialization techniques: *partial evaluation* [JON 96] and *program slicing* [TIP 94, REP 96], as well as a complementary technique that we will call *program fission and fusion*.

- *Partial Evaluation*: A partial evaluator is a program that transforms a program by compiling away the computations based on *static* (i.e. known) information and reconstructing the remaining *dynamic* computations to form the specialized program. This technique has been studied mainly in a functional, logical, or imperative setting, but has also lately been applied to the object paradigm [BRA 00, SCH 00].

- *Program Slicing*: It is a method for automatically decomposing programs by analyzing their data and control flow. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form that still produces the behavior. For instance, this technique can be applied to determine the statements affecting a parti-

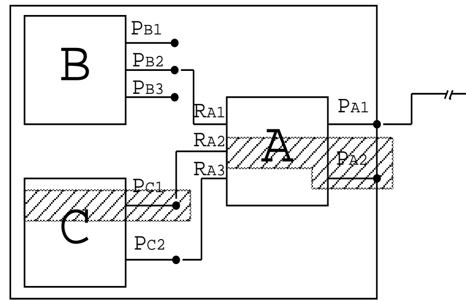

Figure 1. *Composition Example*

cular output statement of a program. Then, if this output statement is no more useful, these statements can be eliminated leading to another kind of specialized program. Initially applied to imperative and functional languages, some work has also been started on object-oriented languages [STE 98].

- *Program Fission and Fusion*: The idea here is to turn some data structure into several data structures that are together equivalent to the initial data structure, and vice-versa. When performing fission, the splitting criteria may be based on the static and dynamic subsets of the data structure. Then, we may propagate just the dynamic subset throughout the program and partially evaluate the computation based on the static subset. This is an extension of *variable splitting* [ROM 90]. Rather than a specialization technique on its own, program fusion and fission should rather be seen as an enabling technique.

3.2. *Scenarios*

In a component setting, each new phase of the life of a component brings new invariants and therefore new specialization opportunities. Here are scenarios illustrating this point.

Firstly, at composition time, it is possible to obtain information about how the components are assembled. In Figure 1, we can see that not all services provided by component A are used in the resulting application. Thinking in terms of specialization, slicing can be performed with respect to the unused output (i.e. provided services) P_{A_2}. After performing specialization on component A, another opportunity is found since the input (i.e. required service) R_{A2} is not required by A any longer. Consequently, a new specialization can be done since the connection between P_{C_1} and R_{A_2} is not required either, and so on.

Secondly, at instantiation time, concrete values for input parameters are given. They can be used to specialize the component through partial evaluation (i.e. by propagating the static input values though the component implementation). In Figure 1

a specialization based on instantiation values is shown. In this case, for example, the provided service P_{B_2} is known to provide a boolean value `true`, then all the computations based on the required service R_{A_1} can be specialized, taking this static value into account.

Finally, specialization can be applied at execution time by determining *quasi-invariants* [PU 95], which are known not to change for some interval of time.

4. Components and Component Generators

In the previous section, we have assumed that program specialization was able to affect the core of a component in an unrestricted manner. How is this possible without breaking the black-box model of the core ? The basic idea is to actually replace the component core by a component generator. Such a component generator can be built at component production time using techniques pioneered by the partial evaluation community around the notion of *generating extension* [JON 93, GLü 95]. One could imagine that such a component generator could generate any possible specialization of the initial component (each specialization corresponding to a given set of invariants). In practice, only a subset of the potential specializations make sense and correspond to planned reuse scenarios. These scenarios can then be used to drive the construction of the component generator, as suggested in [LEM 02] in the context of a specific C-based component model. These scenarios can then become part of the component interface in order to encourage a smart reuse of the components [BüC 97] but also to automatically trigger specialization when connecting components (actually component generators) together [SCH 99].

5. Conclusion

We have shown that adaptability is a key component feature and proposed a basic component definition taking this feature into account. This has hopefully made clear that component adaptation brings many optimization opportunities but that taking advantage of the opportunities associated to the component core calls for sophisticated program specialization techniques. We are currently working on a component model, language and infrastructure based on these ideas.

6. Bibliographie

[BüC 97] BÜCHI M., WECK W., « A Plea for Grey-Box Components », Technical Report n° 122, août 1997, Turku Centre for Computer Science, Turku.

[BRA 00] BRAUX M., NOYÉ J., « Towards partially evaluating reflection in Java », *2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, Boston, MA, USA, janvier 2000, ACM Press, p. 2-11.

[COU 01] COUNCILL B., HEINEMAN G., « Definition of a Software Component and Its Elements », HEINEMAN G., COUNCILL W., Eds., *Component-Based Software Engineering – Putting the Pieces Together*, p. 5-19, Addison-Wesley, 2001.

[DEM 00] DEMICHIEL L., YALÇINALP L., KRISHNAN S., « Enterprise JavaBeans ^{TM} Specification », SUN Microsystems, septembre 2000, Version 2.0, Public Draft 2.

[GAM 94] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[GLü 95] GLÜCK R., JØRGENSEN J., « Efficient Multi-level Generating Extensions for Program Specialization », HERMENEGILDO M., DOAITSE SWIERSTRA S., Eds., *Proceedings of the 7th International Symposium on Programming Language Implementation and Logic Programming*, n° 982 Lecture Notes in Computer Science, Utrecht, The Netherlands, septembre 1995, p. 259–278.

[JON 93] JONES N., GOMARD C., SESTOFT P., *Partial Evaluation and Automatic Program Generation*, International Series in Computer Science, Prentice Hall, 1993.

[JON 96] JONES N., « An Introduction to Partial Evaluation », *ACM Computing Surveys*, vol. 28, n° 3, 1996, p. 480-503.

[LEM 02] LE MEUR A., CONSEL C., ESCRIG B., « An Environment for Building Customizable Software Components », *IFIP/ACM Working Conference - Component Deployment*, Berlin, Germany, juin 2002, Springer-Verlag, p. 1-14.

[PU 95] PU C., AUTREY T., BLACK A., CONSEL C., COWAN C., INOUYE J., KETHANA L., WALPOLE J., ZHANG K., « Optimistic Incremental Specialization: Streamlining a Commercial Operating System », *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, ACM Press, décembre 1995, p. 314–324.

[REP 96] REPS T., TURNIDGE T., « Program Specialization via Program Slicing », DANVY O., GLÜCK R., THIEMANN P., Eds., *Partial Evaluation, International Seminar, Dagstuhl Castle*, vol. 1110 de *Lecture Notes in Computer Science*, Springer-Verlag, février 1996, p. 409-429.

[ROM 90] ROMANENKO S., « Arity Raiser and its Use in Program Specialization », JONES N., Ed., *ESOP'90 - Third European Symposium on Programming*, vol. 432 de *Lecture Notes in Computer Science*, Copenhagen, Denmark, mai 1990, Springer-Verlag, p. 340-360.

[SCH 99] SCHULTZ U., « Black-Box Program Specialization », *ECOOP Workshops*, vol. 1743 de *Lecture Notes in Computer Science*, Springer-Verlag, 1999.

[SCH 00] SCHULTZ U., « Object-Oriented Software Engineering Using Partial Evaluation », PhD thesis, Université de Rennes I, décembre 2000.

[SES 00] SESSIONS R., *COM+ and the battle for the Middle Tier*, Wiley, 2000.

[SHA 96] SHAW M., GARLAN D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[SNY 86] SNYDER A., « Encapsulation and inheritance in object-oriented programming languages », *Conference proceedings on Object-oriented programming systems, languages and applications*, Portland, OR, USA, 1986, ACM Press, p. 38-45.

[STE 98] STEINDL C., « Intermodular Slicing of Object-Oriented Programs », *Proceedings of the 8th Workshop for PhD Students in Object-Oriented Systems ECOOP'98*, Brussels, Belgium, juillet 1998, Springer-Verlag.

[TIP 94] TIP F., « A Survey of Program Slicing Techniques », rapport n° CS-R9438, 1994, CWI.