Published in Technical Report 17/99, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby: Proceedings of WCOP'99

Black-Box Program Specialization

Ulrik Pagh Schultz
Compose Group, IRISA, France
ups@irisa.fr

Abstract

Software components offer numerous advantages in terms of development, but may give rise to inefficiency due to highly generic implementations. Program specialization can automatically remove overheads due to overly generic implementations, but requires inspection of the source code. To retain the advantages of black-box software components, we propose black-box program specialization: specialization opportunities are specified by the developer and become part of the interface of a component, allowing the user to specialize the component according to specific needs.

We characterize the problem of consolidating program specialization and black-box component re-use, outline our proposed solution, and illustrate our approach using an example.

1 Introduction

Software components increase the re-use of existing program fragments, allowing an application to be created by combining existing components. Unlike objects which may implement part functionalities only, a component usually encompasses a complete functionality and functions in an autonomous fashion. In the context of this paper, we consider that components are first created by the component programmer, then either combined with other components by the application programmer (to form complete applications), or inserted into running applications by the end user.

The more a component is generic in its functionality, the more places it can be used, and the greater is the productivity boost that results from code-reuse. However, genericity is known to introduce run-time overheads. For example, parameters that control the behavior of a component may be repeatedly tested even though they remain fixed throughout the lifetime of the component. Also, features that may be needed only in specific situations must be included in a component, even if they are never needed in most applications.

To improve performance, a software component can be manually specialized to obtain a version dedicated to a given application, but such manual albeit systematic modification of source code is error-prone and introduces maintenance overheads. Automatic program specialization is a technique for removing overheads that are due to genericity. Program specialization can automatically instantiate a generic component for a specific context, thus obtaining an efficient implementation. Nevertheless, for program specialization to be effective, it requires guidance by an application programmer familiar with the complete program being specialized, seemingly contradicting the philosophy behind black-box re-use of software components.

As a solution, we propose to extend the description of a component with information regarding the opportunities for specialization that are intrinsic to the component. The burden of identifying the specialization opportunities is placed on the creator of the component, who is assumed to be familiar with its source code. The existing opportunities for specialization are thus externalized, and published as part of the interface of the component. This interface not only allows the application programmer to configure the component to a specific functionality, but also for the framework that joins the components together to automatically complete the configuration process by adapting the components to one another. This con-

figuration process results in a program optimized for the opportunities associated with each actual component.

Overview: First, Section 2 provides an introduction to program specialization. Then, Section 3 describes the main topic of this paper: black-box program specialization. Afterwards, Section 4 describes how components can be automatically adapted when they are installed into a framework. Section 5 provides a concrete example. Finally, Section 6 discusses both the questions raised by this position statement and future work.

2 Program Specialization

Program specialization is the optimization of a program (or a program fragment) based on information about the context in which it is used. For an object-oriented program, specialization can be applied to individual methods of a class. Context information can include both constant values, such as values of parameters, local fields, or fields of other objects, and type information.

One approach to automatic program specialization is *partial evaluation*, which performs aggressive inter-procedural constant propagation (of all data types), and performs constant folding and control-flow simplifications based on this information. Partial evaluation has been extensively investigated for functional [3, 4], logic [6], and imperative [1, 2, 5] languages. This technique has been recently extended to Java by Schultz *et al.* [7]. Partial evaluation is a two-stage process: first the program is analyzed with an abstract domain of values (values are either known or unknown), and then it is specialized with concrete values. While analysis must be performed before compilation, the actual specialization to concrete values can be done at any time. Specialized code can thus be dynamically adapted to situations that have been abstractly analyzed before compilation.

For specialization to be effective, it must only be applied to the performance-critical regions of the program. The application programmer is often aware of where such regions exist in a program, perhaps with the help of profile information. But to optimize such regions, knowledge of how they work is still required for program specialization to be effective. Thus, some understanding of how the source program works is necessary for applying program specialization.

To specialize a software component, it must not only be known how it has been implemented, but also how it will be used. This makes it difficult for the developer to specialize the component, since the exact needs of the application programmer and even the end user must be known. Specialization can be done to cover the most common cases, but it is often impractical, if not impossible, to pre-specialize components for all possible values.

3 Black-Box Program Specialization

To exploit the full potential of program specialization in the field of software components, we must unify seemingly contradictory needs: specialization must be done according to the needs of the application programmer and it must be done at the component level without requiring the programmer to inspect the source code of the component. The solution is to make the available specialization opportunities external to the component. We thus separate the process of declaring specialization opportunities and the subsequent choice of which opportunities to exploit.

Concretely, we propose to express abstract specialization opportunities as part of the interface of a component. The component developer identifies specialization opportunities and the application programmer chooses which opportunities to exploit. To actually perform specialization, we can use a technique such as specialization classes [8], which allow many specialized behaviors to be added to a component, while keeping the original, generic behavior.

Indeed, specialization classes, provide a simple, declarative framework for specifying specialization opportunities and controlling specialization, and might seem appropriate for defining the specialization interface of a component. However, specialization classes work at the implementation level and not at the component level: they

declare opportunities on a class-by-class basis, specifying invariants over values that are private to some object. Also, several specialization classes may be needed to express specialization for a specific opportunity. While useful for actually performing specialization, we conclude that specialization classes are too fine-grained and implementation-specific for specifying specialization opportunities at the component level.

In the same way that the interface of a component must be rich enough to support those operations that are required by the programmer, the set of available specialization opportunities must be rich enough to enable optimization for those cases that are needed. But while an interface that is too limited can hinder appropriate use of a component, a limited set of specialization opportunities will only result in reduced performance. Restricting the programmer to predefined opportunities has an advantage in terms of predictability of specialization: it ensures that specializing according to these opportunities will be beneficial.

The specialization interface should not only allow configuration by the programmer, but also automatic configuration, as described in the next section. Also, the concrete interface that one would want to use will depend on what framework is being used. In Section 5 we provide a concrete example that illustrates what can be needed from a interface, but for now we will leave the definition of the specialization interface as future work.

4 Framework-Based Specialization

Large applications are constructed from many components, so manually performing complete configuration of each individual component would be a tedious task. Furthermore, individual component configuration is only feasible during application development: we cannot require the end user of an application to configure components when they are dynamically installed into the application framework. While individual configuration of a component is essential at application construction time, we deem that it is not sufficient by itself.

Rather than requiring the application programmer to specify how each component should be specialized for a concrete application, specialization can be done based on information about how the components have been combined. This information can be extracted from the components by the framework that binds them together, and used to automatically specialize each component to its context.

A component normally specifies its interface and its dependencies on other components. Not only can the interface of the component include information regarding the specialization opportunities intrinsic to the component: dependencies on other components express more opportunities for specialization. Specifically, dependencies can include specialization opportunities for other components that are due to the way they will be used by this component.

Globally, each component expresses specialization opportunities that influence the specialization opportunities of other components. This can be implemented in a straight forward fashion by allowing each component to configure those components that it depends upon. This allows the components of an application to automatically adapt to the way they will be used. By implementing the propagation of specialization opportunities in the framework, propagation can be done either statically at compile time when the application is created, or dynamically at run time when new components are plugged into the application. Once the configuration process is complete, specialization can be done for the resulting configuration options.

Concretely, the components and their dependencies form a graph structure, and specialization opportunities are propagated through this graph structure until a fixed point is reached. To ensure termination and a simple way of merging configurations, a partial ordering over the possible configuration options could be defined, with the empty set of configuration options as the bottom element and the set of all configuration options together as the top element. (Note that some configuration options may be incompatible; it is the job of the component programmer to decide what happens in the case of conflicts.) Once completely

configured, each component can then be specialized according to the selected specialization opportunities. Formalization and implementation of this approach is future work.

5 Example: Image-Processing Java Beans

As a simple example, consider an image processing application that loads an image, removes noise or performs blurring, and displays the result. When programmed using common object-oriented abstraction mechanisms, such applications can contain many opportunities for specialization [7]. For example, specialization can be done for the concrete representation of image data used while processing the image, or for the actual filters being applied to the image.

In this example, the components will be implemented as Java Beans. The Java Beans framework already encompasses the notion of *properties* that allow configuration of specific parameters. However, this interface alone is too simple for our purposes, since we also work with dependencies between components. Thus, we choose to slightly extend the Java Beans framework, and have a specific mechanism for specialization operations.

Inspired by the BeanInfo interface for complex introspective operations on Beans, we provide each bean with a method for obtaining its *specialization controller* object. This object is the interface performing configuration and the actual specialization operations, as well as describing dependencies between beans to allow automatic configuration. The specialization controller object can be accessed by a BeanBox-like tool for configuration during creation of an application, or by some run-time framework for Java Beans that would permit the specialization controllers to be connected as defined by dependencies between components.

To facilitate the construction of different image processing applications, we program each specific functionality as a separate component. In the context of this example, we are interested in the four following components:

GIFSource: Reads an image from a GIF file.

Median: Removes noise in an image, using a given mask (different kinds of noise are best removed with different masks).

Blur: Performs blurring of an image, using any mask size (the larger the mask, the stronger the blurring effect.)

Display: Displays an image on the screen.

The components work on image data through a generic interface, manipulating image data stored either pixel by pixel or separately for each color.

The blurring algorithm can be specialized for the size of the blurring mask, so the component developer has specified this as a specialization opportunity. In normal use of the application, only a small blurring effect is desired, so the application programmer can configure the `Blur` component to be specialized in advance for a couple of small mask sizes, and to be prepared for being specialized at run time for larger mask sizes. Similarly, the median component can be specialized for a concrete mask.

There is a significant overhead associated with accessing all image data through a generic interface. The components could all be specialized for the same data representation, thus optimizing the program. If the application programmer specifies that the local machine uses packed pixels when displaying data on the screen, the `Display` component can specify this as a specialization opportunity, thus causing all the components that it depends upon to be specialized for this representation.

By a combination of manual configuration and automatic propagation of specialization opportunities, we can thus essentially specialize the graphical application as was done by Schultz *et al.*, which was shown to result in a speedup in execution time between one and a half and five times. However, by specifying specialization opportunities at the component level as was done here, the process of performing specialization has been greatly simplified.

6 Open Questions and Future Work

This position statement outlines an overall approach to applying program specialization to software components. However, many open questions remain:

- Does component-level configuration of each component combined with automatic propagation of specialization opportunities answer some of the needs for specialization of the software component community?
- Is a software component in fact an appropriate unit for specifying possible specialization opportunities and subsequently relying on automatic propagation of opportunities to determine which opportunities to exploit?
- Is it possible to unify the specialization needs that arise when the same component in a single application is used by several different components, possibly each through a unique interface?

In terms of future work, the specialization interface should be defined, and automatic propagation of specialization opportunities should be formally specified. Both should be experimentally implemented in a framework. To minimize the amount of configuration by the programmer, it would be possible to allow components to have "preferences" for certain configuration options. In the example of the previous section, a specific representation could have been the preferred one, and the choice made automatically in the case where the programmer had no preferences (or at run time). Finally, an interface to individual component configuration is needed before this technique can be tested. We envision having a complete graphical interface for controlling and monitoring both approaches, in the form of a *specialization wizard* that can guide the application programmer through the specialization process. In the specific case of Java Beans, it would be advantageous to integrate such an interface with an existing BeanBox tool.

Acknowledgments

This position statement is based on work being done in the Compose group at IRISA. In particular, it is influenced by the current efforts of Phillipe Boinot, Charles Consel, Renaud Marlet, Miguel de Miguel, Gilles Muller, and the author at applying program specialization to software components. Thanks are due to Julia Lawall for proofreading and helpful comments.

References

[1] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[2] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Orlando, FL, USA, June 1994. Technical Report 94/9, University of Melbourne, Australia.

[3] A. Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

[4] C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.

[5] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.

[6] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic program-

ming. *Journal of Logic Programming*, 11:217–242, 1991.

[7] U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 367–390, Lisbon, Portugal, June 1999.

[8] E.N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings*, pages 286–300, Atlanta, USA, October 1997. ACM Press.