Generic Software Component Configuration Via Partial Evaluation

Anne-Françoise Le Meur, Charles Consel

Generic Software Component Configuration Via Partial Evaluation

Anne-Françoise Le Meur and Charles Consel

Compose Group, http://www.irisa.fr/compose
IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France
LABRI, 351 cours de la Libération, 33405 Talence Cedex, France
{lemeur,consel}@irisa.fr

Abstract. We propose the use of partial evaluation to automatically and systematically configure software components in a predictable way. We base our approach on a declaration language that enables the component developer to describe the genericity and configurability of the application components. This information is checked and used to derive components specialized with respect to a given context.

1 Introduction

Modern approaches to developing software systems usually rely on highly generic and re-usable software components. The use of reusable components has a positive impact on productivity, however it is often detrimental to system performance. This inefficiency is rooted at two different levels. First, a generic component must anticipate many usage contexts. To handle a variety of contexts, it may contain much more code than a component designed to provide only a specific functionality in a single context. Second, the separation of an implementation into a collection of collaborating generic components implies the need for communication between these components. Indeed, computation often traverses software connectors, and substantial execution time may be dedicated to gluing components together rather than spent in the components themselves. Overall, both sources of inefficiency are a direct consequence of the fact that genericity is not only present at the design level but also in the implementation.

One way to overcome the efficiency issue is to manually specialize the code. Manual specialization, however, is often not systematic, is error prone, and does not scale up well. These issues directly conflict with typical software engineering concerns, and thus manual specialization can not be considered as a satisfactory solution. However, there exists an alternative to manual optimization, which is automatic program specialization. One approach to automatic program specialization is partial evaluation [5,7].

1.1 What is partial evaluation?

Partial evaluation (PE) enables one to obtain significant optimization by specializing programs with respect to invariants that become valid at various stages of a program's lifetime. Thus, PE systematically maps generic programs into efficient ones, as both execution time and code size may be reduced. A simple example is the specialization of the procedure `dotproduct`, defined in Figure 1(a). This procedure can be specialized for a given array size, say `3`. The result of such specialization is shown in Figure 1(b).

PE usually consists of two phases. First a preprocessing phase propagates an abstract description of its input throughout the program. This description indicates which parts of the input will be available during specialization. Available input part is said to be *static*, whereas unavailable input is said to be *dynamic*. The preprocessing phase annotates each program construct as static or dynamic. A construct is said to be static if it only depends on static input and dynamic if it may depend on dynamic input. The second phase of PE performs the actual specialization. Given the specialization values and an annotated program, the specialization phase evaluates the static constructs and rebuilds the dynamic constructs.

PE has been studied for functional [3,4], logic [9], and imperative languages [1,2,6]. It has reached a mature state, enabling the implementation of partial evaluators for real-size languages such as C [6] and Java [17]. Partial evaluators have been used for a large variety of realistic applications in domains such as operating systems [11], scientific algorithms [8], graphics programs [12] and software engineering [10,15,16]. However, PE has never been integrated within the software engineering process itself.

```
// dotproduct.h
  struct vector{int size; int* values};

// dotproduct.c
int dotproduct(vector* v1,vector* v2) {
  int i;
  int sum = 0;
  for (i=0; i<v1.size; i++)
     sum += v1.values[i] * v2.values[i];
  return sum;
}
```

(a) Generic dotproduct

```
// dotproduct.spe.c
int dotproduct(vector* v1, vector* v2) {

   return v1.values[0]*v2.values[0] +
          v1.values[1] * v2.values[1] +
          v1.values[2] * v2.values[2];
}
```

(b) Specialized dotproduct with v1.size = 3

Fig. 1. Example of program specialization

1.2 How should PE contribute to solving the component configuration problem?

As mentioned, a program application is the combination of several generic components; each component autonomously provides a complete functionality. Typically, to make each component generic, the component programmer introduces several parameters and some code to dispatch on specific behavior and features. Thus, for example, parameters that control the behavior of a component are repeatedly tested even thought they may remain fixed during the execution of the application. These situations are responsible for the performance loss.

In some cases a smart compiler could optimize the code. For example, something like:

```
if(sizeof(int) == sizeof(long))  treat_long(); else treat_short();
```

would be optimized and reduced to a single call. On the other hand, a compiler is totally ineffective in the following case:

```
if (action == ENCODE) encode(); else decode();
```

A compiler is unable to make any optimization if the value of `action` is a program input. However, a partial evaluator precisely targets this situation; this kind of optimization should be a very promising technology for configuring software components.

1.3 Why is it not obvious how to use PE in this context?

Although applying partial evaluation to configure software components to a specific context seems to be easy in principle, it is however non-obvious in practice, for the following reasons.

- Before using a partial evaluator one must first identify the performance-critical code fragments of the program. This can typically be done by the component programmer, who is aware of the component implementation. However, to specialize a software component, one must also know how it will be used. The programmer who uses a software component is often not the one who wrote it. Thus, specializing the component requires the component user to inspect the implementation in details in order to identify the critical pieces of code to optimize. Expecting a programmer to scan the entire implementation of each component, and this for each configuration context, contradicts software engineering principles and defeats the purpose of component reuse. This expertise must be regained each time the application is assigned to a new programmer, modified, or integrated within a larger application. To the best of our knowledge, there is no support to describe this genericity, neither in any programming language nor in any interface definition language.
- Traditionally, program structure follows a functional decomposition. However, this decomposition may not be adequate for applying specialization. For example, code fragments to be specialized may be spread across various functional units. Such fragments need to be manually extracted and assembled to form the program to be specialized. This program must be provided to the partial evaluator along with some configuration and context information. Currently there exists no support to make this process automatic and systematic.
- Finally, even if the critical fragments of code to be specialized have been extracted and the partial evaluator has been configured, there is no guarantee that the resulting program will correspond to an optimized configuration of the program application. This is the direct consequence of the fact that there is no support provided to the programmer to declare clearly what needs to be specialized and thus no support to verify that

the partial evaluator produces a result that matches the programmer's requirements. Configuring software components via partial evaluation has thus so far been an unpredictable process.

2 Our Approach

We believe that PE could be integrated within a software engineering process to configure software components during the development of program applications. However, to reach this goal we need to provide some support to allow a software system to be systematically and efficiently instantiated in a predictable way. To do so, we propose a declaration language aimed at expressing genericity and configurability of a component. For a given configuration, these declarations are verified and used to derive a specialized implementation. In this section, we describe what information, at the component and program level, should be expressed with our declaration language, and explain how the declarations are used to derive the configured components.

2.1 Declaration language at the component level

A fundamental requirement is that component specialization can be done without requiring the component user to examine the source code. Since the component programmer is aware of the critical fragments of code to be optimized and knows what component configurations are possible, he is the best suited to declare what are the specialization opportunities and how the component should be used to obtain a successful specialization. Thus the component developer should provide both the implementation of a component and a specialization interface that describes all the valid configurations and usage contexts.

We have designed a declaration language that allows the component writer to explicitly specify a specialization scenario for each component. A specialization scenario consists of two aspects: a list of the code fragments and data structures of interest for specialization, which may be spread over several files, and an indication of a specialization behavior (e.g. static or dynamic) for each of these elements.

We illustrate this notion of specialization scenario with the following example. Consider a component that contains the procedure `dotproduct` of the previous section. We have seen that one way to configure this component is to specialize the procedure `dotproduct` for a given array size, say `3`. Another valid scenario is to configure it for a known vector v1. In the context of our example, one would write:

```
Module myComponent {        /* Key: Static, Dynamic */

          ...

          Defines {

          From /mylibs/dotproduct.h {

           cfgVect1 struct vector{int size; int* vec};

           cfgVect2 struct vector{int size; int* vec}; }

          From /mylibs/dotproduct.c {

           cfg1 dotproduct(cfgVect1(vector)* v1,  cfgVect1(vector)* v2);

           cfg2 dotproduct(cfgVect2(vector)* v1,  cfgVect1(vector)* v2); }

          }

          ...}

          ...

}
```

This *specialization module* can be read as follows: the component `myComponent` provides a structure `vector` which is defined in the file `/mylibs/mycomponent.h`. The module specifies two specialization scenarios

for this structure, named `cfgVect1` and `cfgVect2`. The scenario `cfgVect1` describes the case where the size of the vector is known but the values are unknown. The scenario `cfgVect2` describes the case where the vector size and the values are known. The module `myComponent` also provides a procedure `dotproduct` implemented in the file `/mylibs/mycomponent.c`. Two scenarios are associated to it, `cfg1` and `cfg2`. In the first case, both arguments of `dotproduct` must satisfy the configuration `cfgVect1`. That is, the size of both vectors must be known. The configuration scenario `cfg2` indicates that the first argument `v1` of `dotproduct` must be entirely known, and the second argument `v2` must have known size and unknown values.

These configuration interfaces can be considered as a means to document the specialization opportunities of a component, making the expertise needed to specialize an application explicit and re-usable.

It is important to note that a configuration interface needs to be associated to a component only if the component presents some optimization opportunities. There is no need to declare a specialization module for components that should be used as is.

2.2 Declaration language at the program level

Large applications are constructed by assembling many components. Thus, our language both declares specialization scenarios at the component level and describes how these components should be composed to form the complete program to be specialized. This composition is defined by building a hierarchy of specialization scenarios.

To define a hierarchy, our language offers constructs to *import* and *export* specialization scenarios. For example, exporting the scenario `cfg1` from the component `myComponent` makes this scenario available for other components to import. Importing a scenario means that we intend to use the associated component in a context corresponding to this scenario.

Exporting a scenario can be seen as associating a *configuration contract* to a given program unit. This contract precisely defines the configuration capabilities of the program unit. Contracts guarantee that specialization of the component will be beneficial if it is used according to the provided scenarios. If one wants to use the component in a context that does not match any proposed scenarios, it is necessary either to carefully extend the component configuration interface to include the new scenario (to ensure the validity of the new scenario, the component code must be inspected) or to use a different implementation of the component.

Furthermore, making explicit these hierarchies of specialization scenarios provides an information that is crucial to systematically configure and optimize each component according to its context, adapting the components to one another in a predictable way, that is, as described in the configuration interfaces.

2.3 Automatic application configuration

Once a hierarchy of specialization scenarios has been specified for a given application, we can configure and optimize all of the components that form this application. This process is depicted in Figure 2. We based our configuration mechanism on a partial evaluator named Tempo [6]. Tempo is a partial evaluator for the C programming language and has been developed by the Compose group.

First, based on the configuration interfaces, the critical fragments of code are extracted from the source files of their various components and composed together to form a program to be specialized. Verification is performed to ensure that the declarations provided in the configuration interfaces are coherent.

Furthermore, because a specialization scenario precisely specifies the programmer's requirements, it can be used to generate the configuration parameters of the partial evaluator. For example, in the case of a partial evaluator for C, specialization declaration context, including aliases, side-effecting procedures, and invariant descriptions. Thus the partial evaluator can be automatically configured based on the module declarations. This is a non-negligible advantage of our approach, as configuring a partial evaluator is a difficult task.

During the next step, the specialization scenarios are checked against the treatments performed during preprocessing phase (as described in the first section) to verify that the configuration contracts are fulfilled. Indeed, it may happen that because of the complexity of an application, the component user has unintentionally used a component in an inappropriate context, implying that specialization will not perform the expected optimizations. Thus, this verification ensures that the specialized program will be configured as described in the configuration interfaces or aborts the specialization process otherwise.

Once all the verifications have been made, the program is finally specialized with respect to the appropriate context values; this results in a residual program consisting in the configured and optimized code of generic application components.

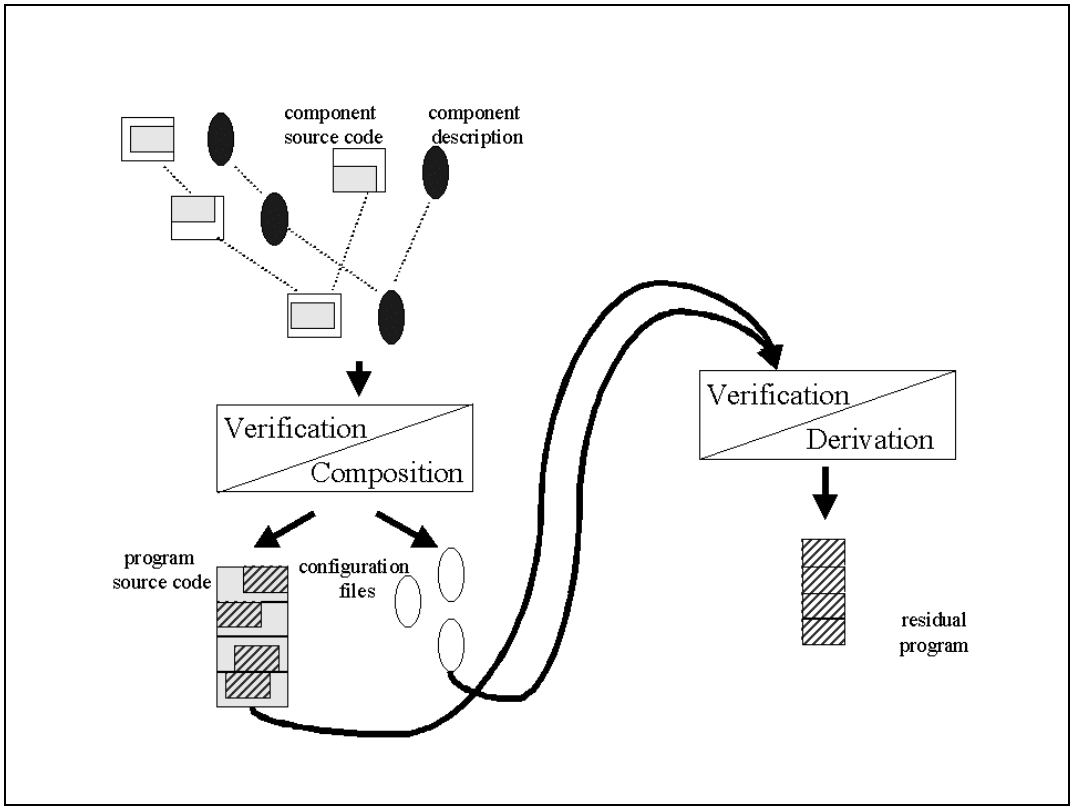

Fig. 2. Abstract view of the configuration process.

3 Conclusion

We propose to integrate PE with the software development process to systematically and efficiently instantiate generic programs. We base our approach on a declaration language that enables a programmer to express the genericity and configurability of a component. For a given configuration, these declarations are checked and used to derive a specialized implementation. Our approach offers the following advantages.

- It provides a high-level declaration language to specify how to configure components and how to combine them. It thus encourages the development of *re-usable specializable components*.
- It allows an automatic and predictable configuration of components. This should considerably improve productivity.
- It does not require the code to be modified. Only the code fragments of interest to specialization are considered. Thus using our approach should not overload the task of the application developers.
- It provides a source of documentation for the configuration capabilities of components.

Acknowledgments

This research is partially funded by the Information Technology for European Advancement -- Project: *Engineering Software Architectures, Processes and Platforms for System-Families*.

The authors would like to thank Julia Lawall for her helpful comments on this paper.

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 and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the twentieth Annual ACM SIFPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493-501, Charleston, SC, USA, January 1993. ACM Press.
6. 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.
7. N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.
8. J.L. Lawall. Faster Fourier transforms via automatic program specialization. In J. Hatcliff, T.AE. Mogensen, and P. Thiemann, editors, *Partial Evaluation: Practice and Theory. Proceedings of the 1998 DIKU International Summerschool, volume 1706 of Lecture Notes in Computer Science*, Copenhagen, Denmark, 1999. Springer-Verlag.
9. J.W Lloyd and J.C Shepherdson. Partial evaluation in logic programming. Journal of Logic Programming, 11:217-242,1991.
10. R. Marlet, S. Thibault, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engineering*, pages 183-192, Lake Tahoe, NV, USA, November 1997. IEEE Computer Society.
11. G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240- 249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.
12. F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. Rapport de recherche 1065, IRISA, Rennes, France, November 1996.
13. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. *In Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 314-324, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.
14. C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. Computing Systems, 1(1):11-32, Winter 1988.
15. S. Thibault and C. Consel. A framework of application generator design. In M. Harandi, editor, *Proceedings of the Symposium on Software Reusability*, pages 131-135, Boston, Massachusetts, USA, May 1997. Software Engineering Notes, 22(3).
16. S. Thibault, R. Marlet, and C. Consel. A domain-specific language for video device drivers: from design to implementation. In Conference on Domain Specific Languages, pages 11-26, Santa Barbara, CA, USA, October 1997. Usenix.
17. 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)*, Lisbon, Portugal, June 1999.