Component Specialization*

Gustavo Bobeff
Ecole des Mines
OBASCO Group - EMN/INRIA
rue Alfred Kastler - 44307
Nantes Cedex 3, France
Gustavo.Bobeff@emn.fr

Jacques Noyé
INRIA
OBASCO Group - EMN/INRIA
rue Alfred Kastler - 44307
Nantes Cedex 3, France
Jacques.Noye@emn.fr

ABSTRACT

Component-Based Software Development (CBSD) is an attractive way to deliver generic executable pieces of program, ready to be reused in many different contexts. Component reuse is based on a black-box model that frees component consumers from diving into implementation details. Adapting a generic component to a particular context of use is then based on a parameterized interface that becomes a specific component wrapper at runtime. This shallow adaptation, which keeps the component implementation unchanged, is a major source of inefficiency. By building on top of well-known specialization techniques, it is possible to take advantage of the genericity of components and adapt their implementation to their usage context without breaking the black-box model. We illustrate these ideas on a simple component model, considering dual specialization techniques, partial evaluation and slicing. A key to not breaking encapsulation is to use *specialization scenarios* extended with *assumptions* on the required services and to package components as *component generators*.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*reuse models*; D.3 [**Programming Languages**]: Processors—*code generation, optimization, translator writing systems and compiler generators*

General Terms

Languages

Keywords

Component-Based Software Development, Partial Evaluation, Program Slicing, Component Generator

*This work was partially funded by the European Commission in the FET Open Domain of the IST Programme under contract no. IST-1999-14191 (EASYCOMP - Easy Composition in Future Generation Component Systems).

PEPM'04, August 24–25, 2004, Verona, Italy.

1. INTRODUCTION

We are at a point in history where software development is changing of scale, switching from programming in the small to programming in the large. Component-Based Software Development [10, 21] is a major element of this trend. The hope is to produce better-quality software faster and with less effort by assembling prefabricated customizable components. Component-based software development relies on clearly distinguishing two roles in the production of software: *component producers* build components *for reuse* while *component consumers* create new applications *by reusing* components. The strong decoupling between producers and consumers relies on making the *implementation* of a component, provided by the producer as a *black box*, out of reach of the consumer. However, in order for the consumer to reuse a component, an *interface* is provided by the producer. The component interface describes the functionality or a set of functionalities *required* or *provided* by the component, commonly known as the component *services*. The role of this interface is to guide composition by rejecting, for instance, incorrect compositions. This role can be seen from two main points of view: a structural point of view (or interconnections) and a behavioral point of view (or interactions) [6, 21]. But, on top of this syntactic role, this interface also has a semantic role, being transformed at composition time (at least conceptually) into a *wrapper*. This wrapper encapsulates the *core* of the component that can be either the related implementation (source code) or the result of the transformation of the implementation (compiled code). Whereas the implementation is a black box, the interface is a *white box*: it can be modified in order to produce a wrapper adapted to the specific usage context of the component.

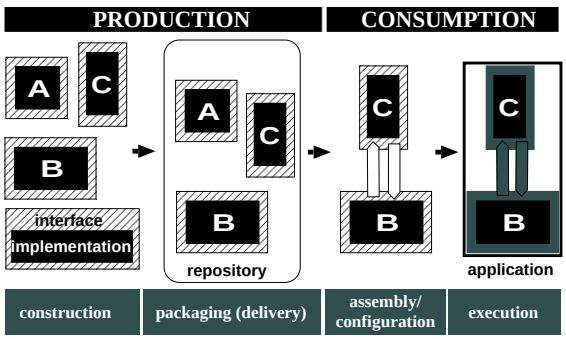

Figure 1: CBSD: Standard approach

This approach, shared by all the industrial component infrastructures (e.g. [7, 20]), is represented in Figure 1. At production time, the component interface, represented by a hatched border, encapsulates the implementation, represented by a black box. At consumption time, the consumer configures and assembles components, using the component interfaces, to build a particular application. In the resulting running application, while the interface is transformed into a component wrapper (represented with a gray border), the implementation remains unchanged (it is still the same initial black box, or a straightforward compilation of it). This makes it possible to adapt components with respect to predefined technical services (persistence, security,...) but only results in a *shallow* adaptation of these components.

In this paper, we revisit and go deeper into the idea evoked in [19] that program specialization can be applied to component implementations while preserving the black-box model of reuse inherent to components. The key is to give to the producer, who has access to the details of the implementation, the means to identify meaningful specialization opportunities and to publish them as part of the component interface. Specifically, we consider a combination of off-line *partial evaluation* [13, 14] and *slicing* [17, 22] to prepare the work at production time and package the components as *component generators*. We suggest to describe the specialization opportunities as *specialization scenarios* [15]. The scenarios must also take the possible usage contexts, here configuration and assembly contexts, into account. As these contexts are not known at production time, this takes the form of *assumptions* on the possible specializations of the services required but not provided by the component, including the availability of configuration data and the usefulness of the services. At production time, the scenarios are used to build the component generator. They are also made part of the component interface. At consumption time, the scenarios are used to select the components, actually component generators, to assemble. The configuration and assembly of these generators triggers the generation of the target application. The generators interact to select the applicable scenarios and generate, in a modular way and without any help from the component consumer, the corresponding specialized implementation code.

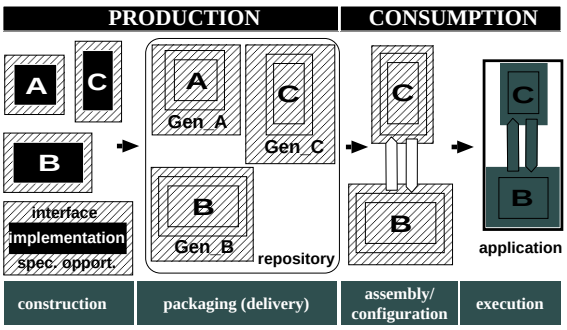

Figure 2: CBSD: an approach based on component generators

To identify clearly the difference between the standard approach and our approach, let us contrast Figure 1 and Figure 2. In the former case, the interfaces are the only source of adaptation (interfaces can be seen as wrapper generators) whereas in the latter case the component generators (entirely hatched boxes) make it possible to extend adaptation to the implementation.

In summary, this paper shows how to integrate a combination of partial evaluation and slicing within component-based software development in order to get a deeper adaptation of software components, and therefore slimmer and faster component-based applications. The whole development life cycle of a component is covered. The key points are the introduction of *assumptions* capturing sensible specialization scenarios to be provided by required services, and the packaging of components as *component generators*. The architecture of a prototype implementing these ideas is sketched.

The rest of this paper is organized as follows. Section 2 introduces the simple component model used for presenting our approach. Based on this model, Section 3 explains how specialization opportunities are described. The generation of the specialized components is described in Section 4. Section 5 discusses related work and Section 6 introduces future work and concludes.

2. COMPONENT MODEL

In order to illustrate our proposal, let us capture the basic features of a component model in the following simple model. We consider a component as an independently deployed unit defined by both an interface representing an explicit contract of required and provided services, and an implementation of the provided services. For the sake of simplicity, we consider that a service has a single entry point. At the interface level, this entry point is represented by a function signature. Figure 3 shows the component *ComputationUnit* designed to perform basic computations typically found in a calculator. The service interfaces are represented as triangle-ended boxes. Required services come into the interface border (e.g. *AdderI* in the component *Multiplier*), while provided services leave the interface border (e.g. *MultiplierI* in the component *Multiplier*). The implementation is presented as a black box inside a component interface (e.g. *MultiplierImpl*).

Component assembly is performed by connecting components, more precisely by connecting provided services of one component to required services of another component. Consumers can build hierarchical architectures by connecting two kinds of components: *primitive* components and *compound* components. A primitive component is built from an interface definition and an implementation block (e.g. *Adder* + *AdderImpl*), while a compound component is built by connecting other primitive or compound components (e.g. *ComputationUnit*). Components included in an enclosing compound component are called *subcomponents* (e.g. *Adder*, *Multiplier* and *Power*). A compound component has also an associated implementation that implements the provided services.

2.1 Component Description Language

Producers and consumers use a component description language (CDL) to describe the architecture of a component-based application as a set of *port interfaces* and *component descriptions*. A port interface is a named set of services grouped according to their intended use. A component description declares the services, required or provided by the component, as *ports*. A port associates an identifier to a

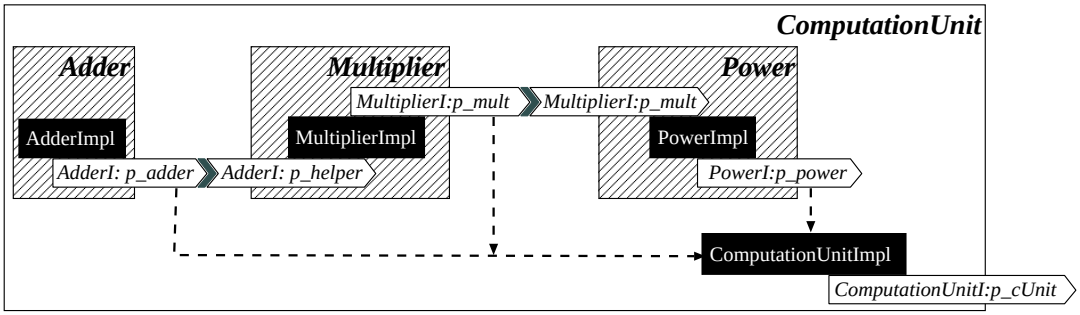

Figure 3: Component Model: *Computation Unit* example

(a) Port Interfaces

```
interface AdderI {
 int add(int x, int y);
}
interface MultiplierI {
 int multiply(int x, int y);
}
interface PowerI {
 int raise(int base, int exp);
}
```

(b) Component descriptions

```
component Adder {
 provides AdderI p_adder;
}
component Multiplier {
 requires AdderI p_helper;
 provides MultiplierI p_mult;
}
component Power {
 requires MultiplierI p_mult;
 provides PowerI p_power;
}
```

Figure 4: CDL: Description of the subcomponents in the component *ComputationUnit*

port interface, which makes it possible to have several occurrences of a single port interface in the same component. The description of a compound component includes the references to its subcomponents, and the *port connections* for connecting subcomponent ports together. For instance, Figure 4 gives the port interfaces and component descriptions corresponding to the subcomponents of the component *ComputationUnit* of Figure 3. In the example, the declaration included in the component description `Multiplier`, shown in Figure 4(b), means that `Multiplier` provides the services specified by the port interface `MultiplierI` under the name `p_mult` and requires the services specified by the port interface `AdderI` under the name `p_helper`. Looking at the description of the compound component *ComputationUnit*, shown in Figure 5, we observe that, in addition to the provided service clause, the component also includes two clauses, `contains` and `connects`. The clause `contains` lists its subcomponents. The clause `connects` declares how the ports of the subcomponents are connected to each other and to ports of the enclosing component. Each subcomponent is an instance of the given component type.

(a) Port Interfaces

```
interface ComputationUnitI {
 int add(int x, int y);
 int subtract(int x, int y);
 int multiply(int x, int y);
 DivResult divide(int x, int y);
 int raise(int base, int exp);
}
```

(b) Component descriptions

```
component ComputationUnit {
 provides ComputationUnitI p_cUnit;

 contains
  Adder adder,
  Multiplier multiplier,
  Power power;

 connects
  adder.p_adder to multiplier.p_helper,
  multiplier.p_mult to power.p_mult;
}
```

Figure 5: CDL: Component description of the component *ComputationUnit*

2.2 Component Implementation

The component description (including port interfaces) and its implementation are associated through a *configuration file* provided by the component producer (see section 4.1). This makes it possible to decouple architectural issues from implementation issues and, in particular, to use different implementation languages. This has to be compared with a language such as ArchJava [3] where a single language is used for the architectural description and the implementation of a component. Moreover, in ArchJava, the description and the implementation cannot be defined separately: the implementation is not a black box any longer. To facili-

tate prototyping, we have however made it easy to choose Java as the implementation language. It suffices to follow simple conventions that can be checked automatically: the port interfaces directly correspond to Java interfaces; there is a Java class, called the *component class*, per component declaration; a component class implements every interface of the provided services; each `requires` declaration adds to the component class a field of the same Java type as the required interface; in a compound component, each provides declaration of each subcomponent adds to the component a field of the same type as the provided interface, etc. The implementation of a compound component can directly use the service provided by its subcomponent. This is represented by dashed lines in Figure 3 (these lines do not belong to the component description).

3. SPECIALIZATION SCENARIOS

The objective of a component producer is to build a component applicable to the widest possible range of contexts (without forgetting that *maximizing reuse minimizes use* [21]). This injects a high dose of genericity into components, which, in the context presented in Section 1, is a major source of inefficiency. For instance, the fact that the implementation is not stripped of its unused services (or part of services) results in fat software. Component configuration (the setting at assembly time of configuration parameters) corresponds to a useless layer of interpretation at runtime... However, as part of the development of a generic component, the producer, who has access to all the details of the component implementation, can identify parts of the implementation that may be optimized under specific conditions. For instance, let us consider a possible implementation of the component *Multiplier* as shown in Figure 6. In the implementation of the service `multiply`, particularly in the conditional and loop structure, we observe that the parameter `x` is used in both the test of the conditional and the test of the loop. If the value of `x` is known at specialization time, then it is possible to specialize the conditional and loop by partial evaluation. Similar reasoning can be applied to the input parameter `y`.

```
class MultiplierImpl implements MultiplierI {
 AdderI p_helper;

 Multiplier(AdderI adder) {
   this.p_helper = adder;
 }

 int multiply(int x, int y) {
  int out = 0;
  if (x >= 0) {
    for (int i = 1; i <= x; i++)
        out = helper.add(out, y);
  } else if (y >= 0) {
    for (int i = 1; i <= y; i++)
        out = helper.add(x, out);
  } else {...}
  return out;
 }
}
```

Figure 6: `Multiplier`: Implementation.

```
specialization MultiplierS
 specializes MultiplierI {
 int multiply(int x, int y) {
  scenario S multiply(S x, S y); //(1)
  scenario S multiply(S x, D y); //(2)
  scenario S multiply(D x, S y); //(3)
  scenario S multiply(D x, D y); //(4)
  scenario D multiply(S x, S y); //(5)
  scenario D multiply(S x, D y); //(6)
  scenario D multiply(D x, S y); //(7)
  scenario D multiply(D x, D y); //(8)
 }
}
```

Figure 7: `Multiplier`: Exhaustive list of specialization scenarios.

Thus, it makes sense that the producer exposes these specialization opportunities to the consumer, in order to provide more efficient alternatives to the generic version of the component. To this end the producer describes such opportunities by defining *specialization scenarios*. This is done by annotating component services with specialization information according to the aforementioned specialization techniques, namely partial evaluation and slicing. A scenario can be seen as a *constraint* defining the dependencies between the input and the output parameters of the component services. For instance, the producer can define a scenario by constraining the parameters with binding-time annotations, that is, static parameters as `S` and dynamic parameters as `D`. For example, `D multiply(D x, S y)` indicates that there exists a specialization opportunity when the input parameters of the service `multiply` are dynamic and static, respectively, while the return value is dynamic. Note that the parameters can be of object type (e.g. the type of return value of service `divide` in Figure 5), then also partially-static declarations can be possible.

As specialization opportunities are strongly tied to a particular implementation, a scenario cannot be defined without taking the corresponding implementation into account. Moreover, the scenarios are only written in terms of services specified in the port interfaces simply because port interfaces are the only information available at assembly time. Indeed, any reference to implementation details would break the black box.

In our context, the producer extends the component description with the specialization scenarios. The producer encloses the scenarios, related to a given port interface, into a *specialization description*. Since the producer does not know the concrete usage contexts of the component, then a possibility would be to include all the combinations on the binding times of the service parameters. For example, the specialization description `MultiplierS`, shown in Figure 7, encloses all the specialization scenarios that can be defined for the component `Multiplier` with respect to the port interface `MultiplierI`. This would mean that the component generator could generate all the possible specializations of the component. Such a systematic policy is not realistic in practice due to potential negative effects on both the time spent at the construction and the size of such a component generator.

To avoid these problems, the producer may reduce the

```
specialization MultiplierS
 specializes MultiplierI {
 int multiply(int x, int y) {
  scenario S multiply(S x, S y); //(1)

  scenario D multiply(S x, D y); //(6)

  scenario D multiply(D x, S y); //(7)
 }
}
```

Figure 8: `Multiplier`: Reduced list of specialization scenarios.

number of specialization scenarios by considering their consistency and benefit in terms of specialization opportunities. Firstly, the scenario `(2)` in Figure 7 is considered inconsistent as we observe in the implementation that it is not possible to return a static value when at least one of the input parameters is dynamic. The same can be said of scenarios `(3)` and `(4)`. Secondly, the scenario `(5)` is redundant with `(1)` as the return parameter can be lifted to dynamic if required. It can therefore be removed (we shall come back to this point in Section 3.1). Thirdly, the scenario `(8)` is also excluded since it does not lead to any kind of specialization. Finally, the producer considers only the scenarios `(1)`, `(6)`, and `(7)` as making sense (see Figure 8).

3.1 Making assumptions

The producer of `Multiplier` does not have access to the implementation of the component that provides the services `add`, consequently she does not know the specialization scenarios available at assembly time. This information would be useful for the producer to define scenarios that maximize specialization opportunities. For instance, in the implementation shown in Figure 6, it would be useful that the service `add` return a static value to obtain a better specialization of the loop structure. Therefore, even though it is not possible to reason in terms of concrete specialization scenarios for the required services, the producer should be able to make *assumptions* about them. For instance, the producer of `Multiplier` can define a scenario `S multiply(S x, S y)` that assumes the availability of the scenario `S add(S x, S y)`. In this case, the assumption only takes into account the return value of the service `add`, given that its input parameters can be inferred from the propagation of the binding times of the input parameters of the service `multiply` throughout the code. One problem here is that assuming at least one static parameter may, if the parameter turns out to be dynamic in the provided scenarios, invalidate the enclosing scenario. For instance, no specialization associated with the scenario `S multiply(S x, Sy)` can take place if the scenario `S add(S x, S y)` is not available, not even specializations that concern computations of the service `multiply` based only on `x` and `y` (e.g. specialization of the conditional test). To avoid this problem, the producer may include other scenarios by relaxing the assumptions. Figure 9 shows the list of specialization scenarios that include assumptions with respect to the required service `add`. The scenario `(1')` is the relaxed version of the scenario `(1)`. Compared to `(1)`, the

```
specialization MultiplierS
 specializes MultiplierI {
 int multiply(int x, int y) {
  scenario S multiply(S x, S y)  //(1)
   in AdderI assumes { S add(S x, S y)};

  scenario D multiply(S x, S y)  //(1')
   in AdderI assumes { D add(S x, S y)};

  scenario D multiply(S x, D y)  //(6)
   in AdderI assumes { D add(S x, D y)};

  scenario D multiply(D x, S y)  //(7)
   in AdderI assumes { D add(D x, S y)};
 }
}
```

Figure 9: `Multiplier`: Specialization scenarios with assumptions.

return value of the service `multiply` is relaxed to dynamic due to the propagation of the return value of the service `add`. For the scenarios `(6)` and `(7)` there is no benefit to make assumptions on the staticness of the return value of the service `add`. In both cases the service call will be residualized because one of the parameters is dynamic.

3.2 Slicing Components

Slicing automatically decomposes a program by analyzing its data and control flow, and the dependencies between its statements [22]. Slicing can be seen as complementing partial evaluation in the sense that *backward* slicing propagates backward information on the output of a program whereas partial evaluation propagates forward information on the input of the program [17]. Actually, the integration of partial evaluation and backward slicing is very natural as soon as partial evaluation is made *use sensitive* [11]. This is further discussed in Section 3.3.1. From a user point of view, a new annotation `K` (for Kill) is introduced. This annotation, to be interpreted as "neither static nor dynamic" is used to define the slicing criterion by telling which part of a service output should be sliced away.

At assembly time, it may turn out that a component provides more services than really needed. A component implementation may not use (or at most partially) some of the services included in a port interface of a required port. In this case, the unused services (or part of them) can be sliced away according to specialization scenarios defined explicitly to guide the slicing process.

Let us consider the service `divide` provided by the component `ComputationUnit` shown in Figure 10. This service not only returns the quotient value, but also returns the remainder value and an error value (i.e. to check the division-by-zero error). Now, let us also consider a target usage context where only the remainder value is useful. In this case, the producer can specify a specialization scenario with the quotient declared as useless while the remainder and error are declared as useful. Consequently, only the computations associated with the calculation of the remainder value will be residualized in the resulting specialized implementation. The description of such a scenario needs to reference the fields of the returning object, as shown in Figure 11.

```
class DivResult {
  int quotient; int remainder; boolean error;
}

class ComputationUnitImpl
 implements ComputationUnitI {
  ...
  int add(int x, int y) {
   return adder.add(x, y);
  }
  int multiply(int x, int y) {
   return multiplier.multiply(x, y);
  }
  int subtract(int x, int y) {
     return this.add(x,this.multiply(-1, y));
  }
  int raise(int base, int exp) {
     return power.raise(base,exp);
  }
  ...
```

```
...
DivResult divide(int x, int y) {
 ...
 if (y != 0) {
  error = false;
  if (x >= 0) {
   if (y > 0) {
    quotient = 0;
    while (x >= y) {
     quotient = this.add(quotient, 1);
     x = this.subtract(x, y);
    }
    remainder = x;
   } else { ... }
  } else { ... }
 } else  error = true;
 return new DivResult(quotient, remainder, error);
}
}
```

Figure 10: `ComputationUnit`: Implementation.

This scenario purely concerns slicing, but it is possible to combine both specialization techniques annotating some of the parameters static. Note also that fully unused services are automatically sliced away as component generators only generate code for used services.

```
specialization ComputationUnitS
 specialize ComputationUnitI {
 DivResult divide(int x, int y) {
  scenario (K quotient, D remainder, D error)
    divide(D x, D y);
 }
}
```

Figure 11: `ComputationUnit`: Specialization scenarios for slicing.

3.3 Defining Scenarios

Both (off-line) partial evaluation and slicing involve complex program analysis. Such an analysis is necessary for identifying specialization opportunities in order to create the scenarios as well as for applying them. Since it is hard and error-prone to do manually we assume that the producer is provided with a proper tool to help with this task, the *analyzer*. This analyzer can also be used to build the assumptions. The analyzer is applied to verify the consistency of scenarios for provided services with respect to assumptions on the required services. It can be used to complete partial scenarios, for instance to derive the binding time of the return value of a provided service from the binding times of its input arguments. Finally, it also helps the producer to assess the benefit of some scenarios and improve the implementation. That is, new or better specialization opportunities may be possible if the implementation (programming style) is modified accordingly.

3.3.1 Integrating Slicing and Partial Evaluation

The integration of partial evaluation and backward slicing is very natural as soon as the binding-time analysis is made *use sensitive* [11].

The idea of use sensitivity is to allow different uses of a given variable definition to have different binding times. In order to do so, binding-time analysis is divided in two steps. A first step propagates forward *use binding times*, corresponding to "standard" binding times with a basic domain $UBt = \{S, D\}$. A second analysis propagates backwards the use binding times computed during the first step in order to determine *definition binding times*. The definition binding time of a location (a variable, an object field...) can be seen as a summary of its uses. It belongs to a new domain defined as the powerset of the UBt domain. The greatest element of the associated lattice $\{S, D\}$ corresponds to a location that has both static and dynamic uses. The smallest element $\{\}$ corresponds to a location that has no use. The definition of such a location can therefore be considered as dead code and the uses involved in this definition ignored, leading in turn to other dead definitions. Backward slicing of a service is then simply obtained by not including in the use summaries the uses corresponding to the part of the return expression to be sliced away.

A drawback of this approach is that slicing takes place after an important part of the binding-time analysis has been done and cannot be used to prune the analysis. Another alternative is therefore to perform slicing first, which corresponds to performing the backward analysis mentioned above on a program with all constructs annotated dynamic. Of course, it can make sense to implement a specific analysis, but this analysis is simply a specialization of the binding-time analysis, no additional specific technology is required.

4. FROM COMPONENTS TO GENERATORS

Packaging components with specialization scenarios is key to making component specialization compatible with a black-box model. Let us first consider a simple case where all the components are implemented using the same language, de-

```
class GenMultiplier extends Generator
 implements GenMultiplierI{
  GenAdderI gen_p_helper;
  String[] listOfScenarios = ...

  GenMultiplier(GenAdderI helper){
   this.gen_p_helper = helper;
  }

  String gen_multiply(String aScenario,
   Hashtable[] staticValues,
   Hashtable[] dynamicParams){
   ...
    // case aScenario do {
    // ...
    // "D multiply(D x, S y)":
    //     return "D_multiply_Dx_3y(x)";
    //     (in the case the static value
    //     of parameter y is 3)
    // }
    ...
  }

  String S_multiply_Sx_Sy(...) {
  ...
  }
  ...
```

```
  ...
  String D_multiply_Sx_Dy(
         Hashtable[] staticValues,
         Hashtable[] dynamicParams){
   String stream;
   int x = ... // bound to the value stored
               // in staticValues
   stream  = "int D_multiply_" + x + "x_Dy";
   stream += "(" + ... unfolding of dynamicParams ...
          + ")";
   stream += "int output = 0";
   if (x >= 0) {
     for(int i=1; i<= x; i++)
      stream += "output = spc_p_helper."
             + gen_p_helper.gen_add(
                              "D add(S x, D y)",
                              staticValues,
                              dynamicParams);
   } else {
     stream += "if (y > 0) { ..."
   }
   stream += "return output;"
   return stream;
  }
 ...
}
```

Figure 12: `GenMultiplier`: Component generator of component `Multiplier` (simplified version).

livered as source code (or even high-level bytecode). We assume that each component consumer has access to the same analyzer and specializer. Then, the assembly can be analyzed and specialized as a whole, with the analysis guided by the specialization scenarios. The main point here is that the consumer does not need to care about the details of the analysis, which simply replays the analysis performed at production time to build and validate the specialization scenarios. An assembly could be rejected because some assumptions cannot be fulfilled, but this would relate to component interfaces available to the consumer.

This case can be refined and its hypotheses relaxed in the following ways. As a first step, each packaged component could include one (or several) annotated version(s) of the implementation, corresponding to the specialization scenarios (the analyzer is no longer needed) as well as a specific specializer. This makes it easier to deal with several implementation languages, although a shared specializer interface still has to be agreed upon. The problem is that this makes packaged components heavy-weight. As a second step, a component can be packaged as a *component generator*. As a first approximation, a component generator takes as input concrete values for the component scenarios and produces the corresponding specialized component. Such a component generator comprises at the same time the initial component, its specialization opportunities, and specialization technology.

Instead of assembling and configuring components per se, a component consumer assembles component generators and provides them with configuration values. Once a concrete usage context has been established, the component generators interact in order to build the specialized application. This interaction involves two phases, *negotiation* and *specialization*. In the first phase, component generators negotiate about the availability of required services (see Section 4.2). This phase finishes when all the requests for provided services have been satisfied throughout the component architecture. In the second phase, the component generators interact to produce the specialized version of services (see Section 4.3), using the information gathered in the negotiation phase.

4.1 Generating Component Generators

Component generation can be interpreted or compiled (there is nothing specific to components here). In the first case, a generic generator builds the component from a binding-time annotated abstract syntax tree [1] and concrete specialization values. In the second case, a specific generator, which can be seen as a specialization of the generic generator with respect to the annotated abstract syntax is generated by a hand-written component generator generator [4], which takes as input the abstract syntax tree. In both cases, the component generator can be compiled to be delivered as compiled code. In the literature, program generators usually produce source code, but producing bytecode or even native code can be envisioned if source code should be hidden to the component consumer for security reasons.

In the following, we assume a component generator generator (CGG) approach but the ideas would apply to the case of a generic generator as well. To illustrate what a component generator looks like, let us take the component implementation `Multiplier` and its specialization scenarios shown in Figure 6 and Figure 9, respectively. The input of the CGG is an annotated version of the services for each specialization scenario, created by the analyzer. The CGG

[1] We are talking about binding times for the sake of simplicity. We actually use an action tree [5].

creates one method, called *service generator*, for each annotated version. A service generator is responsible for generating a specialized version of the service based on the usage context passed through its parameters. The CGG uses the binding-time information to define the code of the service generator as follows: static constucts are simply copied into the body of such a method since it should be evaluated when the service generator is run, whereas dynamic constructs are included as strings in order to be residualized when the service generator is run. The set of the service generators are collected into a class that represents the implementation of the component generator, for example the class `GenMultiplier` in Figure 12.

Each component generator relies on some other component generators to generate specialized versions of the required services. For example, an instance of `GenMultiplier` will rely on an instance of `GenAdderI`, `gen_p_helper` in Figure 12, to provide a proper version of the `AdderI` service. Both interfaces `GenAdderI` and `GenMultiplierI`, shown in Figure 13, define a method (`gen_add` and `gen_multiply`, respectively) called here the *service generator dispatcher*. They are generated automatically at production time as explined in Section 4.4. The component generator class implements this method to provide the signature of the specialized version of a given service, based on the specialization scenarios.

4.2 Negotiation between Component Generators

The question here is what happens if a component generator asks another generator for a service generator that does not exist because the corresponding scenario was not considered at production time? This can be fixed if there exists a smaller scenario with respect to the partial order obtained by extending the partial order defined on binding times ($\sqsubseteq$ with $S \sqsubset D$) to scenarios, using the usual covariant rule on the return value and contravariant rule on the arguments. Similarly to subtyping, a scenario S_1 can replace a scenario S_2 if $S_1 \sqsubseteq S_2$, i.e. if the arguments of S_1 are equal or more dynamic than the arguments of S_2 and the return value of S_1 is equal or more static than the return value of S_2. The selection of a concrete scenario for replacing an unavailable assumed scenario tries to maximize specialization by selecting one of the greatest concrete scenarios smaller than the assumed one, i.e. the one with the most static arguments with respect to the assumed scenario. Binding-time mismatches between a static assumed argument and a dynamic provided argument, or between a dynamic assumed return value and a static provided return value, are solved by *lifting* static values to dynamic. It is also possible to replace a `K` in the return parameter by an `S` or a `D`.

```
interface GenAdderI {
 String gen_add(String aScenario,
        Hashtable[] staticValues,
        Hashtable[] dynamicParams);
}

interface GenMultiplierI {
 String gen_multiply((String aScenario,
        Hashtable[] staticValues,
        Hashtable[] dynamicParams);
}
```

Figure 13: Generator interfaces

```
class AdderImpl implements AdderI {
 int add(int x, int y) {
  return x + y;
 }
}

specialization AdderS specializes AdderI {
 int add( int x, int y) {
  scenario S add (S x, S y);
 }
}
```

Figure 14: `Adder`: Implementation and Scenarios.

To illustrate this situation, let us consider the implementation of `Adder` and the specialization scenario in Figure 14. Figure 15 shows the negotiation between the component generators `GenAdder` and `GenMultiplier`. The component generator uses the method negotiate inherited from the class `Generator`. This method implements the algorithm of scenario substitution described above. Note that both generators include one additional scenario, `D add (D,D)` and `D multiply(D,D)`, respectively. This scenario, called *identity*, is included by the GCC. The systematic inclusion of this identity scenario guarantees that component generation will always be possible (by systematically generating generic components). The interaction takes place as follows: `(1)` given a usage context associated with the scenario `D multiply(S,D)`, `(2)` the selection algorithm checks whether the assumed scenario `D add(S,D)` is provided by the component `Add`. Since no scenario satisfies these constraints, the algorithm searches for a smaller scenario and selects `D add(D,D) (3)`. The binding-time mismatch on the argument `x` is solved by lifting the parameter `x` of the service `add` when called from the service `multiply`. In this case, the scenario `D add(D,D)` satisfies the assumption `(4)`, and the scenario initially required from `GenMultiplier` can be provided `(5)`.

Due to the ordering established between scenarios, there is no scenario smaller than the scenario `S add(S,S)`, even the identity scenario, `D add(D,D)`. A smaller scenario should keep a static return value and have at least one dynamic argument, but those scenarios are rejected by the analyzer (see Section 4). What happens then when the scenario `S add(S,S)` is required, for instance by the scenario `S multiply(S,S)`, and it is not actually provided by `Adder`? As there is no smaller scenario, the specialization must *backtrack* and find an alternative to `S multiply(S,S)`. As there is no smaller scenario either, it is then up to the caller of `GenMultiplier` to consider an alternative. An issue here is to organize the search in order to explore the most static scenarios first. When two or more provided scenarios are smaller than an assumed one, a possible heuristic consists of taking into account the number of static parameters, including the return value, in order to improve specialization. In the case where this number is the same, the selection is done based on the position of the first static parame-

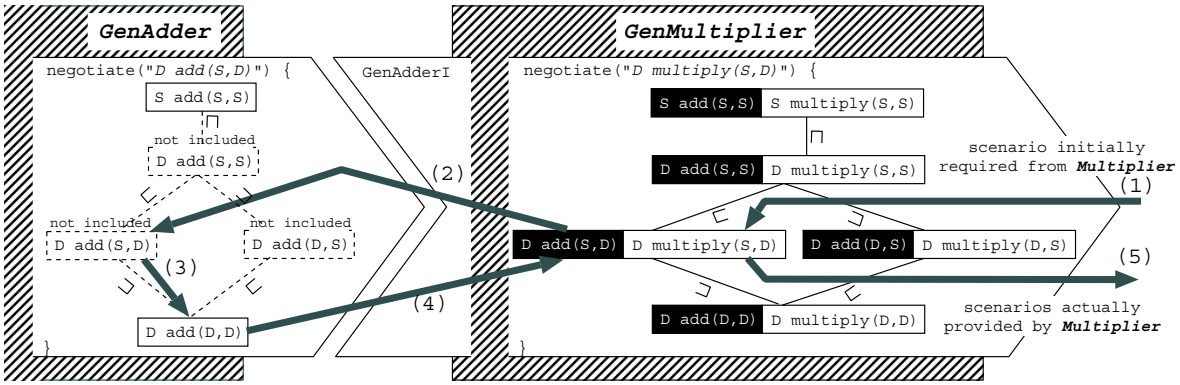

Figure 15: Negotiation process.

ter. For instance, if the search space includes the scenarios `D add(S x, D y)` and `D add(D x, S y)`, the fist scenario is selected. The quantification of the specialization associated with each service parameters is considered as future work.

4.3 Performing the Specialization

To illustrate how the specialization phase takes place, we consider the case where the component `ComputationUnit` is used to translate numbers from the binary to the decimal system. This operation can be implemented as the addition of powers of 2. It then makes sense to consider a specialization scenario for the service `raise` (see Figure 10) since the `base` parameter is static. The computation of the service `raise` provided by the component `ComputationUnit` is actually performed by the service `raise` of component `Power`. For the sake of illustration we also consider the implementation and the specialization scenarios of the component `Power` as shown in Figure 16.

```
class Power implements PowerI {
 int raise(int base, int exp) {
  int output = 0;
  if (exp == 0) return 1;
  if (exp == 1) return base;
  for (int i=1; i < exp, i++){
    output = output
             + multiplier.multiply(base,base);
  }
  return output;
 }
}

specialization PowerS specialize PowerI {
 int raise( int base, int exp) {
  scenario S raise (S base, S exp)
   in MultiplierI assumes S multiply(S x, S y);
...
  scenario S raise (D base, S exp)
   in MultiplierI assumes D multiply(D x, D y);
 }
}
```

Figure 16: `Power`: Implementation and Scenarios.

Firstly, the consumer of the component `ComputationUnit` selects a specialization scenarios and gives concrete values for the static parameters, in our example `D raise(S base, D exp)` and `base = 2`, respectively. Secondly, as described in Section 4.2, the component generators negotiate to know whether the assumed scenarios are available. Figure 17 shows the negotiation between the generators `GenAdder`, `GenMultiplier`, `GenPower`, and `GenComputationUnit`. In this example, no backtracking is needed. This is because all the assumed scenarios have been either included in the required services, or replaced by a smaller one. Thirdly, the specialization of the involved services takes place. The component generator creates a class that contains the specialized versions of the services by propagating the concrete values. Figure 18 shows the resulting specialization for the component `ComputationUnit`.

For the time being, the initial structure of a compound component remains in the specialized component. This can change in the future when component fusion is introduced (see Section 6). But, by default, the interfaces are not retained. This may sometime be an issue as far as external interfaces are concerned. For instance, in the case where a specialized component is generated in order to replace a generic version of the component. Similarly to the *specialization classes* approach presented by Volanschi [24], an *adapter* is then needed to relate the published interfaces to the specialized implementations of the services. Since a single service description may relate to several specialized versions, the adapter can be seen as a switch that selects the proper specialized version based on the usage context.

4.4 Development Tool

Based on the idea presented in this paper, we are currently working on the implementation of a development tool depicted in Figure 19. This development tool interacts with the producer and consumer through a *GUI* implemented as a plug-in integrated to the programming environment Eclipse [1]. At production time, the producer builds a component by writing the component description, the implementation and the specialization scenarios. Also, she can select component generators from the *Repository* to build compound components. This information is written in a configuration file, as mentioned in Section 2.2. Figure 20 shows the configuration file of the compound component `ComputationUnit`. This gives the name of the generator class, the name of the package where the component generator will be stored in, which of the classes involved in the

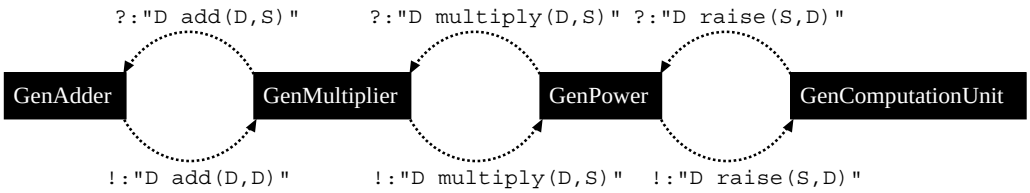

Figure 17: `ComputationUnit`: Scenario negotiation

```
class SpecAdder {
 int D_add_Dx_Dy(int x, int y) {
  return x + y;
 }
}

 class SpecPower {
 SpecMultiplier specMultiplier;
 ...
 int D_raise_2base_Dexp(int exp) {
  int output = 0;
  if (exp == 0) return 1;
  if (exp == 1) return 2;
  for(int i = 1; i <= exp, i++)
   output = output
          + specMultiplier.D_multiply_2x_Dy(2);
  return output;
 }
}
```

```
class SpecMultiplier {
 SpecAdder specAdder;
 ...
 int D_multiply_2x_Dy(int y) {
  int output = 0;
  output = specAdder.D_add_Dx_Dy(output, y);
  output = specAdder.D_add_Dx_Dy(output, y);
  return output;
 }
}

class SpecComputationUnit {
 SpecAdder specAdder;
 SpecMultiplier specMultiplier;
 SpecPowed specPower;
 ...
 int D_raise_2base_Dexp(int exp) {
  return D_raise_2base_Dexp(exp);
 }
 ...
}
```

Figure 18: `ComputationUnit`: Specialization result.

implementation is the component class, the names and locations of the packages that contain the component generator of the subcomponents, etc. The CDL compliance of the configuration file is verified by the *CDL Processor*. The *CDL Processor* also checks that the component implementation conforms to the component description. Once the component information has been checked, the *Analyzer* analyzes (and annotates) the component implementation using the information expressed in the specialization scenarios. In our case, the analyzer relies on a constraint-based approach, and uses REQS [2] as the constraint solver. The annotated component implementation is taken by the component generator generator in order to produce the corresponding component generator. The component generators are stored in component repositories, which are handled by a *Repository manager*. At consumption time, the consumer chooses a component generator from a component repository and provides the usage context by telling which of the provided scenarios will be used and by giving the corresponding specialization values. The *Coordinator* is responsible for triggering the specialization and retrieving the data produced by the different component generators.

5. RELATED WORK

Black-box specialization was first proposed by Schultz in [19]. The basic assumption of this work is the same as ours: a component is a black box. In order to be able to specialize a component, its producer should have specified specialization opportunities and published them in the component interface. Some implementation ideas are sketched in the context of the JavaBeans, where we can find tracks of some of the solutions we have presented here. In particular, the propagation of configuration information and specialization opportunities can be related to the propagation of assumptions in our proposal, and the notion of conflict detection during the propagation can be related to scenario substitution.

The module-based language proposed by Le Meur et al. [16] allows a developer to declare specialization scenarios associated with procedures of modules defined in a target program written in C. Even though a scenario may depend on scenarios defined in different specialization modules, the notion of assumption does not exist. Instead, a *specialization module compiler* verifies the correct referencing of scenarios across specialization modules.

The specialization techniques described here have been combined in a component setting, based on an extension on the CORBA Component Model (CCM), by Hatcliff et al. [8]. Various forms of slicing on the component dependency graph can be used to reason about the system design. A form of partial evaluation is also used to obtain projections of the CCM designs [9]. Specialization is applied at another level

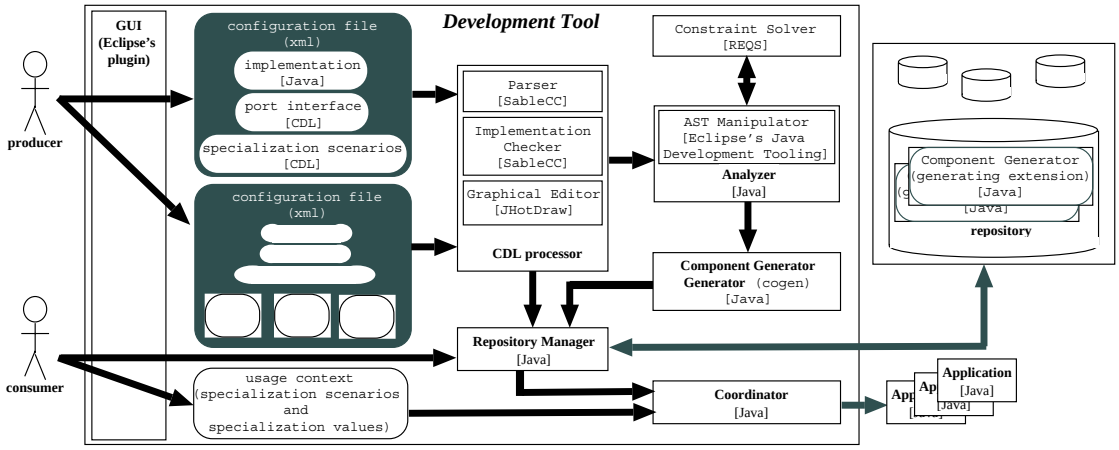

Figure 19: Development Tool

```
<component name="ComputationUnit"
           file="computationUnit.cdl"
           generatorName="GenComputationUnit"
           output="genComputationUnit.jar">
  <implementation main="ComputationUnit.java">
   <source>ComputationUnit.java</source>
   <source>ComputationUnitI.java</source>
  </implementation>
   <subcomponent name="adder"
                 file="genAdder.jar"
                 location=".">
    <provides port="p_adder" type="AdderI.java">
   </subcomponent>
   <subcomponent name="multiplier"
                 file="genMultiplier.jar"
                 location=".">
    <requires port="p_helper" type="AdderI.java">
    <provides port="p_mult" type="Multiplier.java">
   </subcomponent>
   <subcomponent name="power"
                 file="genPower.jar"
                 location=".">
   ...
    <specialization name="ComputationUnitS"
                    file="computationUnitS.cdl"/>
   ...
</component>
```

Figure 20: `ComputationUnit`: Configuration file.

of component development.

A limited form of partial evaluation has also been used for configuration purposes in Koala [23], a component model dedicated to product lines. But specialization is only applied superficially to the component connectors.

Parameterised contracts, introduced by Reussner in [18], map required and provided services of components. They can be used to compute the functionality a component really needs or the functionality provided by the component that is really used. This approach is presented as a generalization of interoperability checks between components in order to enhance the component reusability. Although parameterised contracts can be applied to dynamically recompute a component interface, the component implementation remains unchanged. However, parameterised contracts could be used to perform component slicing at the implementation level since they gather information about the requires-provides relations.

6. CONCLUSION

This work is at an early stage and much remains to be done. We are currently working on a first version of the prototype making it possible to experiment with the whole process and more interesting applications. This first version is based on a minimal component implementation language, namely Featherweight Java [12], with a fairly straightforward binding-time analysis. In particular, we do not consider class polyvariance (all the instances of the same class have the same binding time). Also, only the services are polyvariant, not the methods within a component implementation. We hope that our architecture, through the use of Eclipse and a contraint-based analysis, will make it possible to improve the quality of the analysis without too much effort.

We expect further work on applications to show us that our current definition of scenarios is not sufficiently structured and should be revised, with the possibility of linking scenarios corresponding to different port interfaces. An interesting way to link scenarios would be to complement port interfaces with protocols describing valid service call sequences. This would bring our simplistic component language closer to a real component language (one may argue that such a language does not exist yet). We have also mentioned the possibility of switching between a black-box model and an architectural view of components. This leads to attaching specialization scenarios to an architecture rather than to a component. This is important in order to deal with Software Product Lines, a major application of CBSD. Finally, dynamically reconfigurable architectures could benefit from runtime specialization.

With respect to basic specialization technology, two issues still to be considered are means to define the quality of a scenario and the introduction of structure (i.e. class or component) fusion. The first point should give the component producer the possibility of at least ordering the different scenarios. This would help improving the quality of an assembly when several specializations can be considered. The second point refers to modifying the structure of the pro-

gram by merging classes (within an atomic component) or components. This would be very useful in order to eliminate indirections due to component wrappers (typically used to extend components with technical services, including remote communication).

To conclude, we have shown that, by building on top of well-known specialization techniques, it is possible to take advantage of the genericity of components and still adapt their implementation to their usage context without breaking their black-box model of reuse. A key to not breaking encapsulation is to use specialization scenarios extended with assumptions on the required services and to package components as component generators. We hope that we have also given some clues that program specialization techniques have a key role to play in Component-Based Software Development with many interesting challenges.

7. ACKNOWLEDGMENTS

The authors would like to thank Julia Lawall and Anne-Françoise Le Meur for their comments on a preliminary version of this work.

8. REFERENCES

[1] http://www.eclipse.org.

[2] http://www.irisa.fr/lande/reqs.

[3] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In B. Magnusson, editor, *Proceedings ECOOP'02*, volume 2374 of *LNCS*, pages 334–367, Malaga, Spain, June 2002. Springer-Verlag.

[4] L. Birkedal and M. Welinder. Hand-writing program generator generators. In M. V. Hermenegildo and J. Penjam, editors, *PLILP*, volume 844 of *LNCS*, pages 198–214. Springer, 1994.

[5] C. Consel and O. Danvy. From interpreting to compiling binding times. In N. Jones, editor, *ESOP'90 - Third European Symposium on Programming*, volume 432 of *LNCS*, Copenhagen, Denmark, May 1990. Springer-Verlag.

[6] B. Councill and G. Heineman. Definition of a software component and its elements. In Heineman and Councill [10], pages 5–19.

[7] L. DeMichiel, L. Yalçinalp, and S. Krishnan. *Enterprise JavaBeans^{TM} Specification*. SUN Microsystems, Aug. 2001. Version 2.0, Final Release.

[8] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the* 25^{th} *International Conference on Software Engineering*, pages 160–173, Portland, Oregon, May 2003. IEEE Computer Society Press.

[9] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, V. Prasad, and R. Robby. Slicing and partial evaluation of CORBA component model designs for avionics systems. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'03), June 2003. Invited Talk.

[10] G. Heineman and W. Councill, editors. *Component-Based Software Engineering – Putting the Pieces Together*. Addison-Wesley, 2001.

[11] L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, volume 1302 of *LNCS*, pages 293–314, Paris, France, Sept. 1997. Springer-Verlag.

[12] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[13] N. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, Sept. 1996.

[14] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993.

[15] A. Le Meur, C. Consel, and B. Escrig. An environment for building customizable software components. In *IFIP/ACM Working Conference - Component Deployment*, pages 1–14, Berlin, Germany, June 2002. Springer-Verlag.

[16] A. Le Meur, J. Lawall, and C. Consel. Specialization scenarios: A pragmatic approach to declaring program specialization. *Higher-Order and Symbolic Computation*, 17:49–92, 2004.

[17] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, volume 1110 of *LNCS*, pages 409–429. Springer-Verlag, Feb. 1996.

[18] R. H. Reussner. Automatic component protocol adaptation with the coconut tool suite. *Future Generation Computer Systems*, 19(5):627–639, 2003.

[19] U. Schultz. Black-box program specialization. In J. Bosch, C. Szyperski, and W. Weck, editors, *Fourth International Workshop on Component-Oriented Programming*, Lisbon, Portugal, June 1999. In conjunction with ECOOP 1999.

[20] R. Sessions. *COM+ and the battle for the Middle Tier*. Wiley, 2000.

[21] C. Szyperski. *Component Software*. Addison-Wesley, 2002. 2nd edition.

[22] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[23] R. van Ommering. Building product populations with software components. In *Proceedings of the 24th International Conference on Software Engineering*, pages 255–265, Orlando, FL, USA, May 2002. ACM Press.

[24] E.-N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*. ACM Press, Oct. 1997. ACM SIGPLAN Notices, 32(10).