Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
2003

Component Generators: Towards Adaptable and Efficient Software Components

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Diego Hernán De Sogos

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoter: Jacques Noyé (Ecole des Mines de Nantes)

To Cintia

Abstract

The notion of Software Component offers to developers a possibility to deliver self-contained executable pieces of program that can be integrated into a system. Using components, the software industry can create a market of interchangeable parts as in many other industries like hardware, automobile, etc. Adaptability is vital in this kind of markets. However, the type of adaptability offered in existing component models relies on customizing components by setting some parameterized values, keeping unchanged the structure and operation of the component. This superficial adaptation of software results in inefficient components. Specialization techniques like partial evaluation or slicing cannot be straightforwardly applied to software component implementations, just because the adaptation is done at consumption time, when implementation is in most cases unavailable.

We propose a software component development process that integrates specialization techniques while preserving a black-box model of composition. In order to do that, we extend the definition of a component to include specialization opportunities as part of its interface. To avoid breaking the black-box model, we replace the delivery of a single component by the delivery of a component generator, which will be in charge of producing specialized versions for each of the published specialization opportunities (this kind of generator can be conceived as a generating extension, widely studied in the field of partial evaluation). Specialization can be automatically triggered when the consumer, instead of connecting components, put component generators together and provides the concrete values for specialization.

This work differentiates from previous efforts in the sense that specializable components can be delivered without determining static dependencies with other components (i.e. required services).

Keywords: software component, adaptation, program specialization, efficiency, partial evaluation, component generator, generating extension, black-box model.

Acknowledgements

Firstly, a big thank you to both my supervisors for their time, effort, support, advice and guidance over this work: Jacques Noyé for his constructive criticism and helpful suggestions; Gustavo Bobeff for helping me in everything and always trusting me.

Thanks to the rest of the students and professors of the Ecole de Mines de Nantes that have permit me to be part of their lives.

Special thanks to Gustavo Bobeff's family, for their solidarity and support of every day.

Thank you to Gustavo Rossi for his support and trust. Thanks to all the staff of excellent people that form the LIFIA.

Thank you to my family for their moral support, and for putting up with me during the whole year. Heartfelt thanks to my mother for always being there when I needed and for their unfailing faith in me. Thank you to Miguel for always trusting us. To my brother, Pablo, for daring to follow his dreams. To Flabio, to hold us always in his heart.

Words cannot express my gratitude to my fiancee, Cintia, who frequently knows me better than I know myself, I could not do anything without her.

I am grateful also to all the other friends who have repeatedly told me that I can do it. A special word of thanks to great friend and colleague Federico Naso, who motivated me to make this Master.

Thanks to all of you.

Diego De Sogos

Nantes, France
August 22, 2003

Table of Contents

List of Figures

Chapter 1

Introduction

How to produce adaptable software components without sacrificing neither efficiency nor adaptation? We try to offer an answer to this problem focusing on components as adaptable software delivered in a real black-box fashion at consumption time but prepared for specialization at production time. There are similar approaches proposed, however, to our knowledge, none of them cover the components perspective we are interested in: they are limited to deal with software components as modules or libraries.

Nowadays, the software industry is devoted to producing highly adaptable (customizable)[1] software systems, usually built from components. We can see a software component as a unit of deployment. Producers deliver components having as basic and fundamental premise reuse, and therefore generic functionality. But these advantages do not come for free. Delivering flexible, easy to adapt, and maintainable pieces of software is typically accompanied by a loss of efficiency. This inefficiency is due to two main characteristics of this kind of architectures. The first one has to do with the adaptation and flexibility: because of the many execution contexts that a component has to manage, it has to anticipate any possible variation and provide the appropriate functionality in such a context. The second one has to do with the component architecture itself: communication between components must follow explicit contracts and computations often traverse component connectors, constant verifications of parameters types, etc. that slow down the whole system in comparison with straight connections between them.

Specialization techniques like partial evaluation [14, 28, 11, 43] and slicing [41, 48] cover this issue but they enforce a strong relationship between the component producer and its consumers. These techniques commonly take a complete system implementation and static information (usually configuration information to work in a specific context) as input and produce a specialized version for the input values.

[1] During the rest of the work, we will refer to the terms adaptation and customization indistinctly.

Nevertheless, when deploying a component, there is not a whole system to specialize, but fragments of it. Efficient specialization can be done by the producer once she knows how her component will be used. However, in a component market, one of the key aspects is that the producer does not know where and how the component will be used. In fact, it is only at component instantiation within a complete application that it can be specialized.

A promising solution is the usage of generating extensions [21]. Basically the idea is as follows: the producer can deliver a generating extension for a given component instead of the component itself. Then, when a consumer combines and applies the generating extension in a concrete situation, it is the generating extension by itself which will produce (once the necessary information has been collected) the final specialized component.

The existing literature about generating extensions [8, 20, 30, 21, 25, 6] considers a generator as an independent piece of software that produces a single final product (unrelated to other software pieces). In terms of components it is not possible to do that, the generating extension is built by the producer and at that time the rest of the components involved is not known, therefore the resulting generating extension cannot operate in a stand-alone way. We can say that a component generating extension is not autonomous, it requires to interact at specialization time with other generating extensions (possibly developed by different producers) in order to generate the final specialization of a component.

1.1 Contribution

This thesis proposes a model that adapts generating extensions so that they can be used as component generators. We focus our work on the construction of a component generator generator, that is, a program that automatically builds a special kind of generating extension from a source component. As far as we know, none of the existing specialization models can be applied to a pure component system, unlike the one proposed in this work. As we argued above, most of the current models for adaptable components fall short of a complete decoupling of production and consumption time in the component life cycle, an important issue when thinking of a component market and the delivery of real black-box components to the user. We follow the position stated in [7] attacking this weakness by combining the adaptation techniques of declaring specialization opportunities as part of the component interface with the idea of component generators forming the core of the delivered final product.

The presented model is not intended to be complete but to offer a minimal core to make it easy to reason about component specialization in the presented context. This work gives a conceptual point of view of the proposed model without neglecting

some details on a possible implementation of it.

We have explored the feasibility of the approach by working on a prototypical version of a code generator generator (one of the core parts of our model). We have used the Eclipse [26] platform which is delivered with a full featured Java [4] integrated development environment (IDE). Eclipse provides a Java development tooling (JDT) that allows users to write, compile and edit programs written in the Java programming language, specially by programmatically manipulating the Java source traversing its abstract syntax tree, one of the main activities tied to the construction of a program generator generators.

Although it is not indispensable, the reader should have some knowledge of Java in order to better understand the cases exposed. Even if our model is conceived in a more abstract way, the material presented in this thesis is tied to the way we have chosen to implement the prototype and almost all the examples as well as any new syntactic construction use a Java-like syntax.

Nevertheless, the reader should note that the present work is only a first step towards a concrete implementation of the proposed model. Its focus is on the concepts as well as the main design issues behind the proposal.

1.2 Standpoint

Let us consider the figure 1.1. The figure shows our point of view about working with *adaptable components*[2]. The difference with other component models relies on the way the components are acquired, combined, and how the specialized version of each of them is obtained. A *specializable component* is able to produce specialized versions of itself according to the context where it will be executed. *Producers* deliver *specializable components* requiring some functionality without specifying which component will provide it (this is the case of producer A in the figure, neither A nor B know each other). The common ground between different producers that permits that their components communicate together are the *published interfaces* (in the figure, there is an interface X, which declares the functionality required by A and provided by B). A *consumer* acquires specializable components produced from different sources and combine them, binding required and provided functionality. Once combined, the consumer runs the specializable component with the customized values as input. The *specialized component* is generated automatically as a result of the combination of the specialization of both A and B.

There are several concepts behind the proposed architecture like the common published interface, component market, their participants (consumers and producers), the

[2]In fact, adaptation is achieved through a specialization of the component to the usage context, according to the possibilities offered by the producer (and its target market).

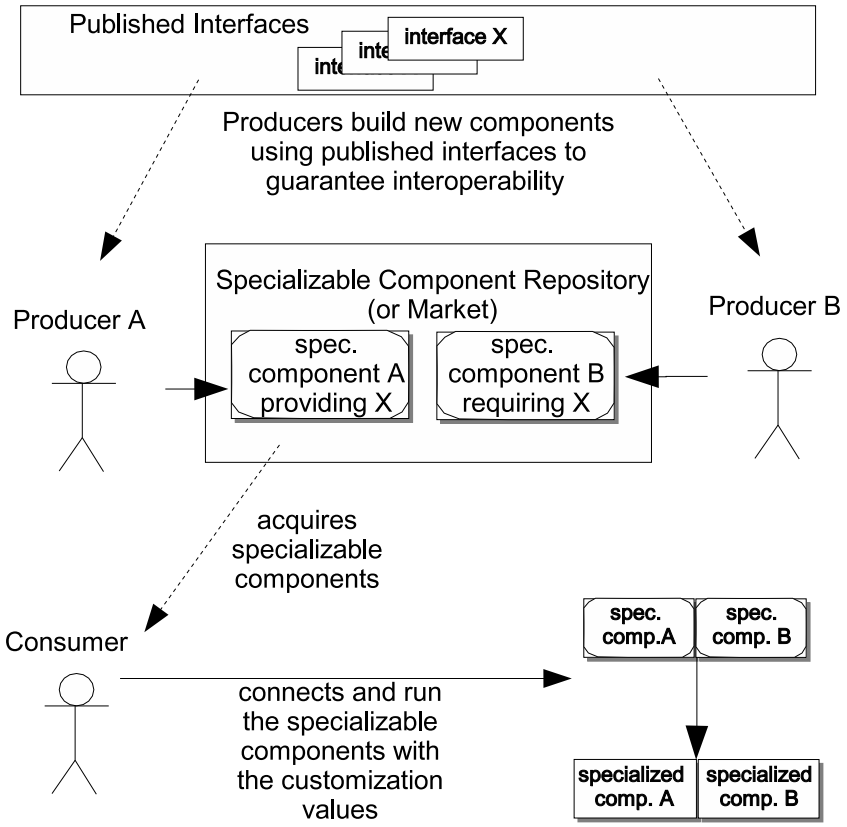

Figure 1.1: Working with self-specializable and independently deployed components

component model (a declaration language and its implementation), the specification of the specialization opportunities, generation of specializable components, the combination and execution processes, etc. Those are the issues that will be addressed and discussed during the rest of this work.

1.3 Overview

This work is organized as follows:

The next three chapters present the main concepts, give background about what components and specialization are about, offering our point of view on terms like components or adaptation. In chapter 2, we introduce the notion of *software components*, and give some definitions used in the rest of the thesis. Chapter 3, states

the problem facing adaptable but efficient software component development. Our solution is related to building *automatized program specializers*, the corresponding concepts and terminology are explained in chapter 4.

The rest of the work is dedicated to explaining our proposal to deal with adaptable and efficient software components. Chapter 5 introduces a basic *component declaration language* that can be used to implement software components in any language, however in our work we have focused our attention to a possible Java implementation of it (in fact, we have chosen Java to show a prototypical version of part of our proposal). The next chapter, 6, introduces *specialization scenarios*, offering a syntactic description explicitly declaring specialization opportunities over a given software component. Chapter 7, about *component generators (CG)*, covers the main concept behind our model. We explain how CGs can be used to obtain adaptable components, we define a language to describe the relationships between them, and we discuss about how they can interact with each other to produce a specialized version of a component. In chapter 8, we present some examples and we also give some guidelines about how a *CG generator* can be built, that is, a program that automatically generates a component generator from a component implementation. We also offer an overview of the whole model, describing briefly the rest of the pieces that are beyond the scope of this work. Finally, we present some details about the prototype of a CG generator that we have made.

Last, chapter 9 comments related work as well as possible future work. A conclusion ends the dissertation.

Chapter 2

Software Components

This chapter defines the term software component and give some definitions about several concepts related to it, such as interfaces, contracts, compositional model, and life cycle.

2.1 Introduction

Software components have become common words in modern software developments. However, their meaning is quite varied. In some cases components are abstractions and appear only at design level, as in most of architectural description languages (ADL). In some development tools, components are graphic user interface (GUI) elements like buttons or windows. We are interested in software components as building blocks of complete software system (not only of GUI aspects), and more precisely in their conception as executable and independently developed pieces of software. In this chapter, we present the concepts around software components widely accepted in the software community, and introduce our point of view of what a software component is about.

2.2 A Background

Historically, development of software products started with building software pieces dedicated to a specific domain and highly tied to it. These pieces presented a monolithic structure that made hard (or impossible) any attempt of reuse (beyond of copying and pasting pieces amongst programs with similar domains).

One of the disadvantages of traditional custom-made software is its maintenance cost and lack of inter-operability in a world of continuous changes (even when the

software was designed for a specific domain, the rapid changes in its environment made obsolete any possible adaptation before it could be productive).

By looking at the hardware industry where the reuse of electronic pieces (prebuilt blocks) had proved to be a very useful technique, it turned out that the same reasoning about creating reusable piece can be applied to the software development helping to deal with the constant development of new software that becomes more complex and bigger day-to-day.

With object-oriented technologies this kind of development is even more natural. The reuse usually practiced within this technology is the reuse of concrete classes. This kind of reuse presupposes implementations (classes) of well-defined and elaborated concepts without making assumptions about collaborations with other classes (that are not part of the same system). One main disadvantage here is that prefabricated code can only be used by new code and not vice versa (the case of legacy systems), which would be very useful in order to make larger structural concepts or frameworks reusable.

With the idea of factoring out as much as possible from the implementations in order to get more reusable classes, abstract classes together with frameworks appeared. Reusable groups of collaborating abstract classes and concrete classes that define a common behavior and structure of various possible applications (within a defined domain) are called application frameworks [50]. One purpose of frameworks is to factor out as much common code as possible into reusable classes, frameworks provide a skeleton that developers can specialize in different ways. It works well as long as the applications share similar structures. However, if we need to alter such a structure it becomes significantly difficult because it is embedded in the framework (the advantage of a provided skeleton becomes a disadvantage from this point of view). Moreover, frameworks are inherently complex, which requires a lot of expertise in order to master them. Even very elaborated frameworks became inflexible when requirements grow along the time. This kind of deficiencies in the object-oriented paradigm lead us to the next step in the research of reusable piece of software: components.

2.3 Component Orientation

A possible definition about what a component is:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Szyperski and Pfister. ECOOP'96, Workshop on Component-Oriented Programming

This definition covers technical aspects like compositional model, interfaces and independent deployment but it also implies the existence of a component market given by third parties composition.

For a component to be independently deployed it has to be well separated from its environment and from other components. This is achieved by defining explicit contracts between the components. We can say that a component can interact (can be combined) with another one, if both respect a published *contract*. Usually, contracts involve the definition of *provided* and *required services*.

We will call *service* a functionality provided or required by a component. It is defined by a set of method, procedure or function signatures (or any other way of modularizing a functionality). For example, a component that work with images could provide the service `Rendering` (with functions like `render(image)`), and it may require a service `Math` (including functions like `multiply(matrix)`).

Explicit context dependencies determine what the deployment environment will require so that the component works properly. Components have a compositional nature, they are units that can be combined following certain rules to obtain more complex units. The explicit context dependencies may include rules of composition (rejecting compositions that satisfy required-provided services but do not obey the rule), deployment, installation and activation of the component. Those context dependencies are part of the component specification.

Because a component must be able to interact in many different contexts it has to provide very general contracts, it must deal with generic provided services. Therefore, a component requiring a service provided by another component has to be prepared to work with any kind (at least a wide variety) of different implementations of those required services.

Figure 2.1 shows how we represent graphically a component. A component is represented by a shadowed box. The outgoing provided services (grey figures in the right side of the box) can be connected to the required services of another component (filling the space outlined by white figures drawn in the left side of the box).

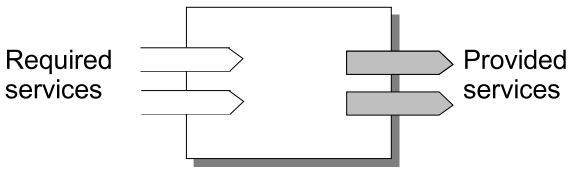

Figure 2.1: Graphical representation of a component

Required and provided services of a component contracts are usually specified with interfaces (see below 2.4.2). An interface specifies a set of named operations with associated semantics: two interfaces may provide an operation of the same name (receiving the same parameters and producing the same type of output), however, they might produce a different value as output because the meaning assigned to the operation is different.

Interfaces do not only connect components but also allow the communication of two different actors in the life of the component, the producer and the consumer. In a component setting, providers and consumers are ignorant of each other, so interfaces that allow proper configurations become the common ground between them.

Producers and consumers are the participants of a component market where interfaces are published and components are delivered and acquired. The success of a component market will depend on how well defined and standardized are the interfaces that allow producers and consumers to use and to compose the components. If a component offers a lot of services, but it provides such services through non standard or well known interfaces, then that component has no market (it will not be used by any third party because it cannot be successfully composed with other components).

2.3.1 Benefits

Software components help to fulfill two key premises of the software engineering process:

- to decrease production costs reusing existing, software
- to enable easier maintenance of delivered systems.

Components have been quite successfully in producing reusable pieces that evolve with time and in dealing with increasing complexity. Component-based software development consists of assembling systems using independent but plugable parts (the components). The key difference between composing classes and component is precisely the independence of the components, a part makes no assumptions about which other part will be plugged into it. Of course, there must be some kind of configuration of the component in order to be able to communicate with the other parts and to get proper interaction between the assembled parts. This generic behavior as well as the capability for configuration (and adaptation) are essentials in a world of components.

The upgrade cycle of software affects any traditional system. Fully integrated systems require periodic upgrades (migration of old databases, new hardware, compatibility between existing pieces of software, etc). Software components replace the maintenance based on changes in software and architecture implementation, by a maintenance based on software evolution. If a new version of a component is available then the system can evolve just by unplugging the old version and re-plugging

the new one (because it is the same component, it must respect established contracts). In some cases, where there is a major evolution of the system, it may be necessary to also modify the architecture. In traditional environments, a change in the architecture might be done implicitly during a change in the implementation, making future maintenance harder. Using components, a change in the architecture is forced to be done in an explicit way, it implies adding different contracts and relationships (otherwise components will not be able to interact each other and the system will stop working), again helping maintenance.

2.3.2 Differences with Objects and Modules

The high-level specification of functionality and encapsulation of the implementation may lead us to think of a component as an object. Components differ from objects in the sense that a component encompass a complete functionality in an autonomous fashion. Their main advantages are genericity (and consequently reusability) and independency. If we also consider that the implementation of a component is completely independent from its declaration and configuration, then we can put components and objects on different levels: components are associated to an architectural view of the system, while objects are strongly associated with the implementation level. Note that a complete decoupling of architecture and implementation could not be very healthy for system evolution, it may allow inconsistencies, causing confusion, violating architectural properties, and making the system hard to understand[1].

Although object-oriented programming is not a prerequisite for components, most of the modern component models are implemented by means of one or more classes. A component encapsulates the implementation and makes the services accessible through public interfaces. Components and objects are orthogonal concepts, and by taking the object-oriented perspective we can benefit of their advantages within the component field.

So far, it seems that components are closer to what modules are. Modules have always been used to package multiple entities, such as abstract data types or indeed classes, into one unit. Unlike classes (which normally cannot be deployed as independent units of execution), modules can be seen as minimal components, for example, a graphic library which provides services implemented in a functional or object-oriented way. However, some of the key aspects of components are adaptability and configuration, which are not allowed with modules. Modules tend to depend statically on implementations of other modules, those imports are hardwired as constants in the code (for example, module inter-dependencies), it makes impossible to adapt it without rebuilding the module: components can be deployed declaring some services as

[1] ArchJava [2] is one of the approaches trying to keep closer architecture specification and implementation.

required, without the need to explicitly hard-code the component that will provide such a service. Obviously, modularity is a desired property and it must be part of what a component is.

2.4 Building Components

Often single object-oriented classes or white-box frameworks are hard to reuse because many internals have to be understood before the white box can be reused. The characteristics of self-contained units with explicit interfaces for connecting elements naturally leads us in a black-box component model approach. This is the view adopted for most of the existing component models like Enterprise JavaBeans [33] or CORBA [5].

Decisions about what should be a component (what things can be encapsulated and treated as a unit) and how it can be built (contract declaration, composition, implementation) involves the following issues:

1. Granularity, how small or big should it be.
2. Interface specification, that is, its provided and required services.
3. Composition model, how to compose components, what are the rules for composing components.
4. Independent extensibility, a deployed unit should be extensible with other units independently deployed.

Next, we explain in more detail each of these issues.

2.4.1 Granularity

What are the elements inside each black-box component? Because in most cases, we will end up with an object-oriented implementation, it can be hard to think about what parts of our object model can be encapsulated as a component. In [47], Szyperski lists some aspects to be taken into account when determining the appropriate granularity of a component such as common reuse, releasability, reusability and local changeability.

To decide what things can be grouped and considered as a component is it necessary to understand better what we are talking about when we say self-contained units. An important aspect of self-contained components is the common reuse that we can do with such a unit: the computational parts present in a component should be used

together. Common reuse is the basic rule to select the granularity for a component. An important part of common reuse is the notion of *coupling* and *cohesion* [37].

A self-contained component should be a *releasable* and *reusable* entity. These issues are related to the granularity as well. We can ensure version compatibility quite easily for a self-contained black-box component, but it is hard for a single class or a complex framework. Therefore, releasability and reusability can help us to decide if some degree of granularity is appropriate or not: if reuse/release equivalence does not hold, we may have chosen the wrong granularity.

Another important aspect is *local changeability*, that is, components should be designed in a way such that changes only apply locally to the component, but are unlikely to affect other components (this is mainly acquired by strictly allowing the communication between components by means of required or provided interfaces). Local changeability is another restriction to find self-contained entities: computational structures forming together a similarity group in their expected changes are good candidates for forming a component.

2.4.2 Interface Specification

Interfaces define the access points of a component. Technically an interface is a set of named operations that can be invoked by clients (typically other components). Normally a component can define several interfaces corresponding to different kinds of services.

The semantics of an operation specification can involve declaration of several elements like type definition, protocol specification, invariants, etc. This information needs to be declared in the interface. It serves both providers to implement the interface and clients to use it.

Because a component and its clients are developed separately, it is the contract provided by the interface that forms a solid basis for a successful interaction.

In an object-oriented implementation, interfaces are specified as method signatures, and the implementation associated is late bound during method invocation. In a component system, it is necessary to introduce more control to ensure the right connectivity of components declared in the architecture of the system: as we say above, interfaces may contain other information beyond a method signature, like invariants or pre and post conditions. With components, most of component composition relies in parts deployed by third parties, designers cannot make assumptions about the components beyond the information published in the interfaces.

In a component setting, interfaces are part of the component contracts declaring what the clients need to do to use it and what the provider has to implement to fulfill the services it includes. In terms of a single operation declaration, these specifications

correspond to pre- and post-conditions for the operation. Clients have to meet the pre-conditions before calling the operation, and providers have to ensure post-conditions before returning to the client.

Having interfaces as contracts between components, it is straightforward to define the substitutability of components: a component A can be safely substituted for another component B, if B provides at least the same interfaces as A and if B requires at most the interfaces required by A.

2.4.3 Compositional Model

Compound components can be built by composing atomic components and other compound components. This distinction should be transparent for clients. This is a very basic principle for building component-based systems. If components are built from other components, they should be usable in the same way as the atomic components in the system. Thus a component concept has to be *scalable* to be usable for larger component frameworks. That is, there should not be different interfaces for using atomic or compound components. A client of a compound component should not have to be aware of the component construction details.

In most of the existing component models, the composition relationship forms a directed, acyclic graph. Cyclic component dependencies mean that all parts, involved in a cycle, have to be released simultaneously. Thus there is a strong dependency in all members of the cycle. Regarding reliability this means that we may end up with a system in which nothing works until everything works [38]. It also means that the developers of different component have to interfere with each other to make changes to their own code. This breaks local changeability[2].

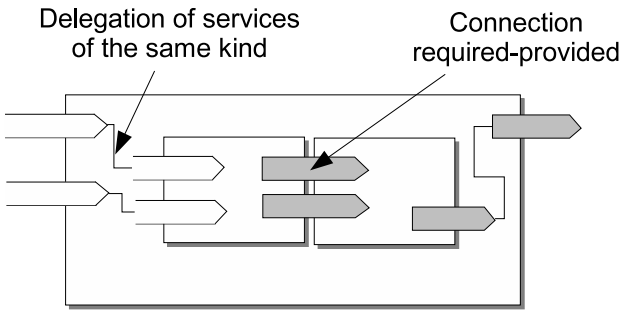

Figure 2.2: A compound component consisting of two subcomponents

[2]However, some recent proposals, like Jiazzi [34], consider the possibility of cyclic references. They use techniques like the open class pattern and mixins to keep the local changeability and to be able to release independently part of the nodes that forms a cyclic graph.

In figure 2.2 we can see how a compound component can be represented graphically. The subcomponents form the internal structure of the component, which is hidden to the user. The enclosing component declares two required services (that are delegated to the left internal subcomponent), and provides one service (the one really provided by the internal component at the right). Lines between services of the same kind denote delegation of the service. Within a compound component, all required or provided services should be connected between them or connected to a service declared in the enclosing component.

2.4.4 Independent Extensibility

A system is independently extensible if it is extensible and if independently developed extensions can be combined [46]. There are many examples of this kind of systems, where the extensions are usually called *plugins* (maybe one of the first examples can be the internet browser Netscape Navigator). Most of the time this kind of systems is one level extensible, in the sense that plugins are not themselves independently extensible. Components take this property as a recursive construction principle: a component accepts an extension that (because it is a component) can accept other extensions and so on.

Thinking this way, we can end up with many small components resulting in a completely partitioned system. Even if it maximizes reuse, it conflicts with efficiency and robustness.

Why performance is affected can be easily answered if we think about how expensive is a cross-context call. Having a system partitioned in plenty of subsystems each of them respecting explicit contracts and consequently constant checking of compatibility and connectivity introduces a prohibitive overhead during service dispatching. Here, the key to extensible, independent, and efficient system can be to choose the right granularity for components. We will go back to the discussion about the problems related to efficient and flexible components in chapter 3.

2.5 Component Life Cycle

Several stages can be recognized in the life cycle of a component [1, 17]; they usually include implementation, deployment (and assembly), acquisition, configuration (adaptation), and run time. Though selection of concrete components for subcomponents is usually done at the assembly stage, in different models, for example when considering third-party components, the component selection may be postponed to later stages, like the configuration for execution, or also at the run-time stage, in case of dynamic environments.

Component deployment means making the component available for reuse. Components are deployed with a specification that typically includes a set of required/provided interfaces (as mentioned before it composes the contract of the component). Sometimes a specification of usage can be necessary (which services can be combined with which others, or a sequence of services that must be followed to obtain the desired functionality).

With respect to the implementation of a component there are two main different alternatives: the first one, where the specification and the implementation are integrated in one single language; and the second one (the model chosen in this work), where the specification is written in a declaration language and the implementation can be performed in any other programming language. In this last approach, components can be implemented by means of one or more classes or even by conventional procedures.

The implementation of a component forms a self-contained unit of execution that can be acquired by a consumer (for example through a global component repository, where each component can be classified by its specification) to combine it with other components to build a final application or to produce a new component (the consumer is the producer of a new component).

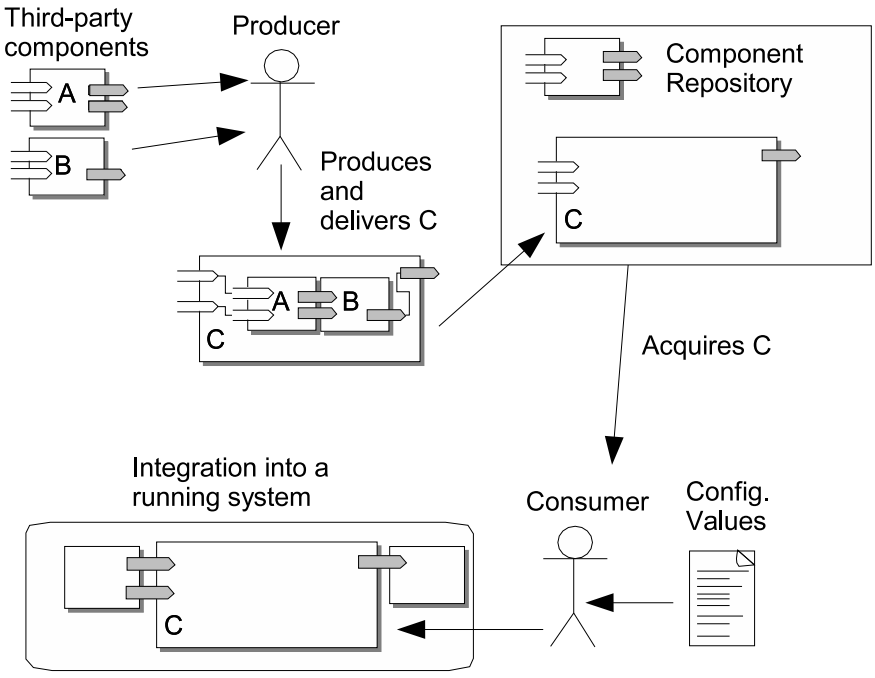

Figure 2.3: Component life cycle

These deployed components come with a certain set of configurable aspects[3], commonly represented by attributes (adaptation information), which are specified at configuration time (when the components are integrated into a running system). Usually, once the configuration attributes are fixed during the component configuration process, they are not allowed to be changed during the rest of the component life. This restriction guarantees future substitutability of a component for another respecting the same contract (2.4.2) (otherwise there may be relationships in the system that could depend on some specific component attribute, because is not part of its configuration it could not be provided by a different component when replacing it).

Configuring and executing components is often done in composition environments by means of scripting and direct linking. We can also consider a top-level component (which is a composition of components) as a program by itself (this top level component encapsulates a given configuration).

So far, we had to distinguish between component configuration attributes and just component attributes. The first ones are concerned with the specification of some adaptation of the component during system configuration time, that is, when components are combined and linked, before runtime. The second ones refer to properties of a component already adapted to a given context and that may change during the system execution, this kind of attributes is related to systems where components are instantiated and consequently different instances can have different internal states. It is a more pragmatic vision of what components are, closer to object-orientation. For some authors, components can only exist at configuration level, as elements describing the architecture of the system; once the system is running, what one really instantiates and executes are just objects. Note that if we want even more flexible system where configuration can be changed on the fly (during system execution), components should survive the system architecture (they must exist as first-class citizens at runtime level), otherwise architecture and implementation will not evolve together making maintenance and system evolution hard (a change of a component property in execution would not be reflected in the component specification).

[3]We are using this word in a general sense here, although a link could be drawn to the aspect-oriented programming.

Chapter 3

Producing Adaptable Software Components

Components enable software industry to produce reusable and easy to maintain elements. Adaptation plays a main role in these aspect of software industry. Most of the time adaptation is obtained by delivering quite generic components that, even when adaptable, become too big as they try to cover all the possibilities. In other cases, adaptation is reached at the cost of increasing execution time. In both cases, the efficiency (in terms of required space or runtime execution) is affected.

In this chapter we discuss about the kind of adaptation we are looking for in a component and why it is hard to obtain efficient and adaptable software components.

3.1 Introduction

Component have to be generic enough to be reusable and therefore to be deliverable to a wide variety of clients (to have a place in the component market), covering many different contexts. Most of the time, a component consumer only wants a few parts of the whole functionality provided by a component (resulting from the needed genericity). In other cases, the customer will use some service but always applied to a same subset of values. Components should provide consumers a way to adapt (or customize) the functionality provided.

We will talk about an adaptable software component as one that uses information provided by a consumer (or indirectly received from a consumer through other components) to configure itself (or through an external tool) in a more convenient way. We say that the component is adaptable to its execution context.

The genericity required in a component leads to build a piece of software prepared to deal with any possible variant. Usually, components deal with adaptation

restricting their provided services to some configuration values. But this adaptation falls short of taking advantage of these values to optimize the implementation of the component, in fact it will contain useless code to handle all the possibilities.

3.2 Degrees of Adaptation

One of the key issues in software industry is reuse. How much a component can be reused depends of the ability of the component to be adapted (customized) to work in different contexts.

Adaptation refers to the ability of a component to satisfy requirements related to the context in which it is used. Most of the modern component architectures offers component consumers some ways of adapting components. For example, JavaBeans [45, 33] offer the possibility to explicitly declare some attributes (called properties) as parameters to customize the software component (called bean). This kind of adaptation relies on setting some execution values and maybe restricting specific behavior. We can call it *functional* or *behavioral* adaptation[1].

Reuse concerns can be satisfied with behavioral adaptation, however it is not a solution to obtain efficient component systems, another important issue within this industry. If your component is highly customizable and can be adapted to almost any environment but its average performance is not so good, you still will have problems to find satisfied consumers.

With behavioral adaptation, the component code remains unchanged. It means that if the component has to be prepared to work in many different contexts, then its code has to be generic enough. The result is that the component spends more time checking contexts and parameter values[2] than doing real work.

A second kind of adaptation can be addressed at code level. *Code* adaptation aims to take advantage of parameterized values by eliminating unnecessary checking of such values. It eliminates the generality and consequently produces smaller and faster programs. However this kind of adaptation is not straightforwardly applicable to components. It has to do mainly with the way components are delivered, but to better understand this problem let us start talking about efficient deployment in a non-component software development.

[1] Do not confuse this kind of adaptation with the one that consists of *changing* the functionality of a component. In this work we are not considering such a kind of adaptation, we focus on how to adapt a provided functionality to work properly (and more efficiently) in a given context.

[2] In terms of programming: conditional clauses, switches, double dispatching or many extra procedure or method calls to solve how to process a service.

3.3 Efficiency–Adaptability Trade-off

When a software is built for a specific context and usage, the software developer can produce an optimal code for it. Even when at design level there can exist some concepts and patterns that are applied in order to obtain a piece of software easy to maintain and to evolve, the final product can be improved in terms of efficient execution, for example using compiler optimizations such as elimination of dead code (because of a known execution context). In terms of traditional software development where the producer and the consumer belong to the same team, there is no problems to combine efficient software with very flexible and adaptable pieces.

This approach works well until the producer decides that she does not want to spend time and money developing a part of its product that was already made (and tested that it works) by other producers (third-party producers). And this is precisely what components are about: pieces of software produced by third-party developers that any other developer can use without caring about the details about how they were built. Using this kind of software implies to obey the contracts published by their developers (this is the way the producer can guarantee that the component will work).

Back to the idea of delivering a final optimal product (in terms of execution and hardware requirements, low execution time, and low memory usage), it is more complicated to get such an efficiency with third-party components. The problem is that third-party components are delivered in a black-box fashion (and that is precisely the advantage, developers do not care about implementation issues, just about functionality), however, to know how to optimize such a component implies to get into details about how it is implemented. An objection here could be: why do not third-party producer already deliver optimized components? The answer lies in that the component producers are creating pieces of software that target a wide range of possible consumers. They even do not know who their clients are and therefore cannot deliver us specialized software. Moreover, opening the black box and performing some kind of optimization over it does not make any sense if we think that these components can be at the same time composed by other third-party development.

On the one hand, a component cannot succeed (in terms of competitive product, if we think about a component market) if it cannot be used efficiently. On the other hand, producing an efficient component implies to reduce its scope of usage, which leads again to a not so useful component.

3.4 Dealing with Efficient Components

Faced to this perspective of component optimization, what are the possibilities?

To improve performance, a component can be *manually* specialized (which relies

on the usage of a white-box model, adapting domain specific behavior by eliminating generic code, for example, that checks for certain conditions, etc). But applying these systematic modifications of source code could lead to a nightmare in a huge system. Moreover, this kind of actions are error-prone and lead to maintenance overheads. As we said before, in terms of software component, it implies that each time a consumer acquires a component it has to contact the producer to tell her about how the component will be used, which is of course unrealistic.

The alternative to manual specialization is *automatic* program specialization [3, 13, 21, 41, 15, 44], a well known technique for removing overheads that are due to genericity[3]. The problem with automatic program specialization is that it is not completely automatic, it needs to be driven by an application developer familiar with the structure of the program being specialized (in this sense we can say that automatic program specialization is a white-box technique). This approach relies on the programmer to code *how* to customize a program. This presents several problems. First, the development process continues to be error prone because the programmer has to take care of both normal computation and code generation; second, testing and debugging tools rarely deal well with programs that generate programs, which results in more responsibility on the programmer; and finally, programs are less readable because they are cluttered with compilation directives and other optimizations for code generation. Moreover, talking specifically about software components, we face the same problem of consumer-producer knowledge cited before.

Black-box program specialization [42] deals with this issue by including specialization opportunities as part of the component interface. In this way the information about what are good specializations can be delivered together with the component to the consumer (this approach emphasizes *what* to optimize instead of *how* to do it). The idea is that once components have been composed and integrated in a system, the configuration is completed with a step of automatic specialization driven by the published adaptation opportunities declared in each component. This mechanism is formalized by Le Meur et al. in a framework to build adaptable software components [31]. However, the position taken by Le Meur relies on a modular composition and declaration of software rather than components. The programmer groups specialization declarations in a specialization module, as the component code is being developed. Once the components and their associated specialization modules have been developed and the consistency between the implementation and the declared specializations have been checked, they are delivered together. A component consumer links the components together and specify the concrete specialization values for each component. Finally a transformation engine takes as input both the adaptable component (the component code and its specialization modules) and the user-provided values, and automatically generates the adapted component ready to be integrated

[3]This technique will be discussed in more detail in the next chapter.

in a complete system. Note that this approach considers a reduced view of components as modules: any required service is fixed at deployment time, once an adaptable component is built by the producer (linking implementation and specialization modules), any dependency with other module is already bound. In fact, the user receives an adaptable module instead of a component, where she cannot decide which other components to connect but only specify some parameterized values. What happens if you want to produce and deliver a component C which provides some service X and requires a given service Y, where the provider of Y cannot be determined during deployment of C (figure 3.1)? This approach is not suitable to produce such components. All required services need to be bound before the specializable component is deployed.

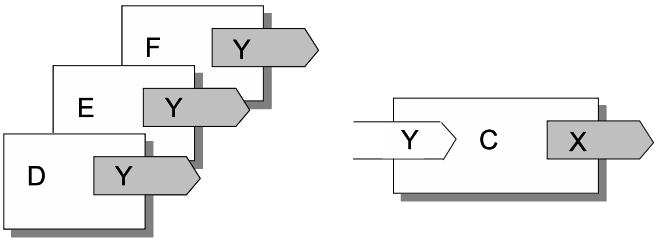

Figure 3.1: No specification of required services during component deployment

This last approach to specialization can be modified to adapt it to a component-based programming model by changing the responsibilities involved in the generation of the specialized component: instead of having a global configuration step that produces the specialized version, we can delegate this responsibility to each component itself, more precisely to what is called the *generating extension* of the component [21, 29], which is basically a dedicated component generator (it produces a specialized version of the initial component[4]). Back to the example of having a component with provided service X and required service Y, we can build the generating extension of this component, which will not fix any specific component for the required Y, but will be able to connect with other generating extension that provides Y as services (that it, whose specialization provides such a service). This is the approach taken in this thesis to deal with this efficiency-adaptation problem. In the following, we will introduce a basic component model and will propose a model that allows the automatic creation of this kind of component generators.

[4]The general concept of generating extension is explained in section 4.3 in next chapter.

Chapter 4

Program Specialization

This chapter presents background information about program specialization, and explains the concepts of generating extensions and program generators generator, which are the basis for our model of adaptable software component.

4.1 Introduction

The purpose of program specialization is to automatically transform a generic program (or program fragment) into a specialized (and hopefully optimized) version of it based on information about the context where the program will be executed [29, 41]. In object-oriented environments, specialization can be carried out at a class level by specializing the methods of the class. The context information can include values of parameters, constant fields, type information or even properties (e.g. the value of integer variable `x` is restricted to positive values). In terms of adaptable components, the context is composed by the adaptation that a component consumer can apply to the component in order to integrate it in a bigger system (probably another component). A component must be prepared to deal with a range of values, but once such values are fixed by the consumer, the generic code introduce unnecessary overhead that can be removed by this technique. We can think about a *program specializer* as a function that takes a generic program P and some context information S (part of input of P, or desired output, etc.) as input and returns a specialized (optimized) implementation of P for such a S context.

The classic example is the specialization of the function `power` (using a functional notation):

```
power n x = if n=1 then x
                   else x * power (n − 1) x
```

We can specialize this function by fixing `n` to the value 3 (in this case, n will be

the static context information S). Which would lead to:

```
power3 x = x * (x * x)
```

Which is obviously a better version of power for those cases, where power is always applied with n fixed to 3.

There are several approaches to program specialization, such as *partial evaluation* [29], *program slicing* [41, 48] and some complementary techniques like *program fission and fusion*. In this dissertation we focus on partial evaluation as the technique used to build generating extensions (therefore, component generators).

4.2 Partial Evaluation

Partial evaluation is one of the specialization technique most widely studied overall in functional [10], logic [32] and imperative [13] environments. In last years there have been some results in object oriented languages such as C++ [49] and Java [43] as well.

Partial evaluation is a special form of program specialization where the context information provided to the specializer consists of part of the input data of the program to be specialized.

A *partial evaluator* is a program that takes a source code program P and part of P input data (sIn, what is called the *static* data), and produces as result a new program P_{sIn} (the residual program), which given finally the rest of the input of P (the *dynamic* data dIn), will be able to produce the same output than the original P.

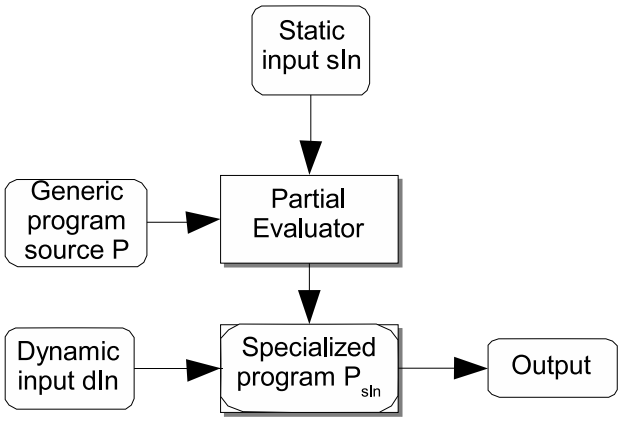

Figure 4.1: A partial evaluator

In figure 4.1, we have represented with rounded boxes the input and output data, and with shadowed boxes the programs. Note that P_{sIn} can be seen as output data from the side of the partial evaluator and as a program itself receiving the dynamic remaining information.

In this way, one could write a highly parameterizable but inefficient program and afterwards use a partial evaluator to automatically obtain an adapted efficient version[1].

4.2.1 Binding-Time Analysis

Partial evaluation is often staged in two phases: analysis and specialization[2]. During analysis, *binding-time* information is collected and such information is used to guide the second specialization phase, which consist of the specialization itself: the partial evaluator takes the static input and the analyzed program and generates a specialized version for that input.

Binding time analysis is the process of determining at what time the value of a variable can be computed, that is, the time when the value can be bound to the variable [29]. We can distinguish two specific binding times: compilation time (static binding time) and execution time (dynamic binding time). Static inputs of a program have by definition static binding time, while the remaining input information of the residual program is classified with dynamic binding time.

Generalizing the concept to nodes of an abstract syntax tree (AST) of the program, we may call *binding time* the degree of availability of some part of a program (a node) during program specialization. A node annotated as *static*, implies that the information necessary to compute such a node(and its corresponding children) will be fixed at specialization time. Computations involving static nodes can be calculated at specialization time and therefore, removed from the final specialized component (the *residual* program). A node annotated as *dynamic*, means that it is part of a computation that involves unknown values at specialization time, this kind of nodes involves a deeper study to determine whether it will be reduced or rebuilt in the residual program: a node is dynamic as soon as one of its sub-nodes is dynamic; depending on its type (if it is a loop, a conditional, a variable or whatever), and the annotation of its sub-nodes, it can be either rebuilt (it will be present in the residual program) or reduced (it will be replaced by a new node in the residual program). Binding-time information is propagated throughout the program structure resulting in a complete annotated program; propagating binding-time information is often made automatically by what is called a binding-time analyzer (BTA). The input

[1]The specialized version can be more efficient due to the fact that the computations depending on static inputs are *compiled away* by the partial evaluator.

[2]This kind of partial evaluators is called *off-line partial evaluator*.

of a BTA analyzer is the static data and the result of a BTA analysis is a binding-time tree structurally identical to the source AST, but annotated with binding-time information.

```
if n=1
 then x
 else
  x * power (n − 1) x
```

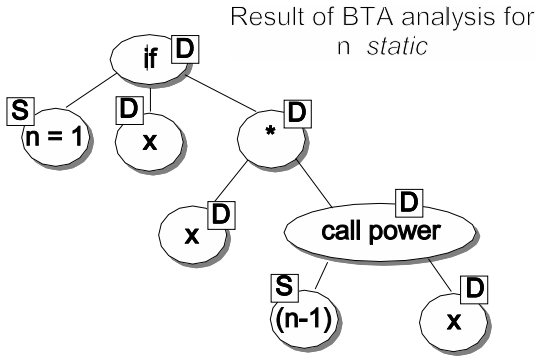

Figure 4.2: Annotated abstract syntax tree for power function

The figure 4.2 shows the annotated AST for the body of function `power`, having made an analysis that starts taken n with static binding time. Nodes labelled with S have static binding time, and nodes labelled with D have dynamic binding time. Note that the node `if` (the root in the example) has dynamic binding-time (because even though the condition can be determined during specialization time, its sub-nodes have dynamic binding-time), however it could be desirable to eliminate the node `if` during specialization (because the condition is static). To handle cases like this, it is possible to add an intermediate step between the BTA and the partial evaluator, which consists of taking advantage of other information like the type of node (in this case a node `if`), and including *annotation action* in the nodes. An annotation action states whether the node must be *reduced* or *rebuilt* (all static nodes will have a *reduce* annotation action, but dynamic nodes may have either reduce or rebuild annotation action, see the nodes labelled as static but with different actions in the figure 4.3).

Binding-time analysis is the first and most important step in the specialization process, it constitutes the base for the rest of the stages. The accuracy of the binding-time information impacts directly over the quality of the specialization. There are many techniques to perform binding-time analysis, such as dataflow analysis or constraint-based analysis [36]. We will not look at the details of BTA. Our objective is to obtain adequate component specializers beyond of what kind of analyzer is used.

```
if n=1
 then x
 else
  x * power (n - 1) x
```

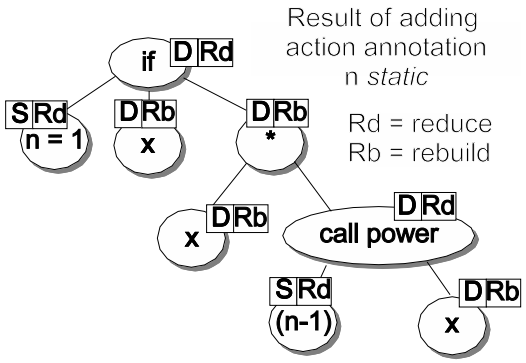

Figure 4.3: Annotated AST complemented with action annotations

4.3 Generating Extensions

When we specialize a program we do that for a group of concrete values that appears in a given situation (the context information used by the specializer). It could be very tedious if we had to build a specialized version of our general program each time a different context appears, even if it can be done automatically using a partial evaluator. Moreover, working with such a general partial evaluator (that takes any component as source) would be very slow. Instead, what we can do is to build specialized versions of this partial evaluator (specializing the specializer) [29].

If we call `P` the original program, and `mix` a partial evaluator that takes `P` and context information `x` as input, we can get the specialized version $\texttt{P}_x$ of `P` for `x` by:

`mix P x =` $\texttt{P}_x$

If `mix` and `P` are written in the same language, we can build the specialized version of `mix` for `P`, called `mixp` by running something like (assuming a functional-like curried function):

```
mixp = mix mix P
```

That is, a partial evaluator that, given some context information `x`, produces a specialized version of the original `P`

If we parameterize the previous program, to be able to use it with any other `P`, what we obtain is what is called a program generator generator (in the literature it is called cogen [19, 25]).

Figure 4.4 (borrowed from [29]) shows this situation. We introduce another level of indirection: the generic program `P` is taken by the `cogen`, which builds a specialized partial evaluator `P-gen` (called the generating extension of `P`). Finally, `P-gen`, receiving the static information behaves like described in figure 4.1.

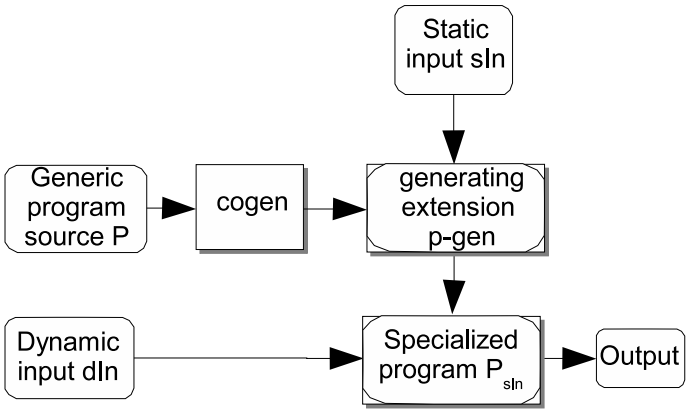

Figure 4.4: A generator of program generators

To obtain such a generator, the so called direct approach (applying `mix` to itself) is unrealistic. We have to deal with problems like double-encoding, huge constructed expressions (or worse, infinite code explosion), unreadable functions, unnecessary tagging and un-tagging operations in the generated program, etc. Instead of using the indirect approach for building generating extension, a more feasible alternative is the idea of handwriting a program generator generator. This idea was presented by Birkedal and Welinder in [6], they showed that building this kind of program not only is possible but it is easier and more efficient than optimized self mix applications. It turns out that generating extensions have a very simple and natural structure. Besides some house keeping code for keeping track of which constructs have been specialized, with which values, they present a structure almost symmetric to the one of the original program: for each function `f` in `P`, there is a function `rd-f` that creates a specialized version of `f` [18]. In the body of `rd-f`, the computations annotated as static remain unchanged (copied as is from `P`), while dynamic ones are replaced by operations to build abstract syntax trees.

Applied to the `power` function mentioned above, the generating extension for `n` static could look like:

```
power n = "powern x = " + rd_power n
rd_power n = if n == 1 then "x"
                       else "x * " + rd_power (n -1)
```

where `+` is assumed to be a primitive to concatenate strings. Variable `n` is static, therefore expression `n = 1` has static binding time and with a static condition, we

can compute the `if` branch during specialization.

This approach is simpler because the cogen only does syntax manipulation of the program's annotated AST.

Chapter 5

A Component Declaration Language

There are two possibilities to program software components: by providing a single implementation language that natively support component declarations (as ArchJava [2] does, extending Java syntax); or by decoupling system architecture (component declaration) from the implementation.

We have chosen the second one, mainly because our aim is to offer a model for building specializable components and we want to keep it sufficiently abstract to be applicable over existing component technologies (like CORBA [22] or JavaBeans [35]). We define the CDL as an architectural description language. The language is intended to be simple and a starting point to reason about component generators. Java is used as the implementation language.

5.1 Introduction

Software Component is an ambiguous term and it has been interpreted in many different ways sometimes contradictories. This chapter defines a terminology and gives a formal syntax to be used as a *component declaration language*. The presented language is not intended to be the ultimate component language but a basic language that allow reasoning about component generators.

The semantics of the component declaration language is informally described by a mapping between the component description and its implementation using Java in section 5.3. Strong and static type checking of Java programs facilitate automatic verification of the implementation with respect to the architecture. Because Java is an object-oriented language, the mapping between component declaration and class implementation can be easily visualized considering that a component is implemented by a single class.

Note that everything related with efficiency and specialization opportunities is left aside in this model. In this way we clearly separated any aspect concerning implementation issues[1] from our component architecture. In particular, we do not need to think about how components will be specialized when defining their architecture. Even during the implementation stage of such a defined architecture, we do not think about *how* but about *what* can be specialized while implementing the component.

5.2 Component Declaration Language (CDL)

Following the definition given in 2.3 we define a *component* as a unit deployed independently and declaring an explicit contract of required/provided functionalities. A component declaration must be attached to an implementation in order to be delivered to the client.

We call *service* a set of logically related functions provided or required by a component[2]. A component can provide or require more than one service, each of those services will be associated to a variable definition (with the service as type) which is called *port*. Declaration of services is given by Java interface declarations [4], we associate the type of a Java interface to a provided or required service. In order to provide a clearer analysis about how components and their specialization interact between them, we forbid passing a component as parameter in a method declaration. In some sense this restriction also helps to keep clean communications between components from an architectural point of view. We always know what components are interacting together by looking at the connection of provided and required services. It ensures that the architecture represented in this language accurately represents communications between components. This idea of clean and explicit communication through component interfaces is known as communication integrity and it is present in other component models like ArchJava.

The *contract* of a component is given by the set of required and provided services (ultimately, a set of interfaces).

A component definition cannot appear inside another component definition. However components can be *compound* (a component can declare variables containing other components, called *subcomponents*). We refer to a *primitive component* as one that does not include any internal reference to another component.

A primitive component declaration only exposes its contract. For compound component, there can be a *connection clause*, which binds or delegates provided and required functionality to their subcomponents. A binding is valid when it connects

[1] We will define a mapping between declaring and implementing components to provide a better idea of our concepts and to be able to associate specialization opportunities with concrete examples.

[2] Function in terms of computational unit, it can be a procedure, a method, a logic predicate...

a provided service with a required service, or when a service declared in the compound component is connected to a service of the same kind (provided or required) of some of its subcomponents(in this case we say that such a binding is a *delegation of service*). Informally, a compound component is *well-defined* (in terms of valid declaration) if for each of its subcomponents, all required services are bound with some proper provided services.

The matching between provided and required services is given by a correspondence of type names in the declaration. If component C provides X, and component D requires Y (even when X and Y denote structurally equivalent services), these services cannot be connected. The usage of structural compatibility remains as a subject of discussion. The problem of structural compatibility is that behind an interface declaration there is associated a semantic meaning. Even when the structure is the same, if this meaning is not, the components will not work properly together.

Components deployed for further utilization by another producer or a consumer only publish part of its CDL specification: the required and provided interfaces. The internal structure (subcomponents and their connections) are not published. On this way, we preserve the encapsulation and we hide implementation details from the client. Note that for a person that acquires a new component, the fact that a component is primitive or compound is transparent (we preserve the characteristics of the object-oriented software deployment).

5.2.1 Syntax

A component specifies its required and provided services declaring the corresponding interfaces for them in the component declaration. Next, we introduce the grammar for the component declaration language using the concepts presented in the previous section.

Conventions:

Program text (keywords and program separators) are in **bold**
Meta-variables are in *italic*

[x] means optional occurrence of x (zero or only one occurrence of x)

x | y means either x or y

Domains:

identifier ∈ Identifier

Syntax:

component_def ::= **component** *component_name* **{**
component_interface
component_structure
}

component_interface ::= [*requires_clause*]
[*provides_clause*]

component_structure ::= [*contains_clause*]
[*connects_clause*]

requires_clause ::= **requires** *requires_list*;

requires_list::= *requires_def* |
requires_def, *requires_list*

requires_def ::= *interface_name input_port*

provides_clause ::= **provides** *provides_list*;

provides_list::= *provides_def* |
provides_def, *provides_list*

provides_def ::= *interface_name output_port*

contains_clause ::= **contains** *variable_def_list*;

variable_def_list ::= *variable_def* |
variable_def, *variable_def_list*

connects_clause ::= **connects** *connects_def_list*;

connects_def_list::= *connects_def* |
connects_def, *connects_def_list*

connects_def ::= *var_input_port_access* **to** *var_output_port_access* |
output_port **to** *var_output_port_access* |
input_port **to** *var_input_port_access*

variable_def ::= *component_name variable*

var_input_port_access ::= *variable*.*input_port*

var_output_port_access ::= *variable*.*output_port*

Auxiliary definitions (only to help comprehension of part of the semantics)

component_name ::= *identifier*

interface_name ::= *identifier*

input_port ::= *identifier*

output_port ::= *identifier*

variable ::= *identifier*

5.2.2 Example

Let us consider the architecture described in figure 5.1 for a component that could be part of a calculator: `ComputationUnit` provides two services, the addition provided ultimately by the `Adder` subcomponent, and the multiplication provided by the `Multiplier` subcomponent.

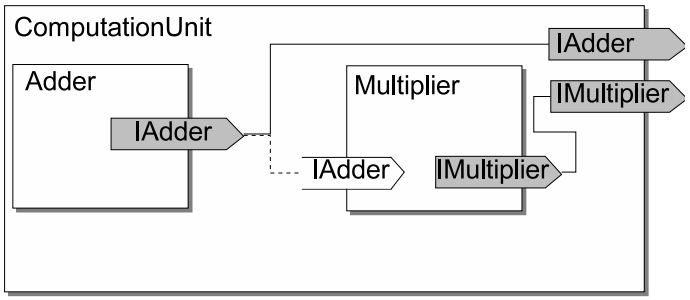

Figure 5.1: A compound component connecting two subcomponents

Internally `Multiplier` requires that somebody provides the service `IAdder`. The component `ComputationUnit` connects the required service with the component `Adder`. Note that the service provided by `Adder` is shared by two components: the enclosing `ComputationUnit` and the subcomponent `Multiplier`.

Moreover, the service `IAdder` is shared in different ways. A filled line indicates the delegation mechanism used with the enclosing component (`ComputationUnit`), while a dashed line indicates a connection of a provided service with a required one, in this case `Adder` with `Multiplier`. This is the reason why we draw in a different way the composition, instead of putting required services with provided services together as we have shown in previous figures.

Using our CDL we can describe this architecture as follows:

```
interface IAdder {
    int add(int x, int y);
}

interface IMultiplier {
    int multiply(int x, int y);
}

component  Adder{
    provides IAdder adder;
}

component Multiplier{
    requires IAdder helper;
    provides IMultiplier  multiplier;
}

component ComputationUnit {
    provides IAdder add, IMultiplier mult;
    contains  Adder adder,   Multiplier  multiplier;
    connects  multiplier.helper to adder.adder,
              mult to multiplier.multiplier,
              add to adder.adder;
}
```

5.2.3 Static Architecture

Having a look at the proposed CDL, the reader can easily identify that it allows only static connection of components. The clause `connect` and `contains` only accept static arguments either a single value or a list. We chose a static environment because it is the simplest scenario to show our principles of component specialization. Slightly relaxing some of the clauses and providing more flexible data structures (like semi-dynamic arrays) could be added to this model without inconveniences, however dynamic structures introduce other complexities at the level of binding-time analysis and code generation that we prefer to ignore in this initial model.

5.2.4 Component Instantiation

Provided and required services are declared using ports, which consist of an interface type and a variable name. In this way, we use such a variable instead of a type when connecting compound components and provided or required services. It not only promotes homogeneity in the connections but also allows reasoning about component instances, for example, the component `ComputationUnit` could declare two internal components of type `Adder`, which means that two different instances of the `Adder` component will exist during system execution. By connecting the variables instead

of the type, we can describe different architectures. Modifying the example shown in figure 5.1, we get the architecture described in figure 5.2.

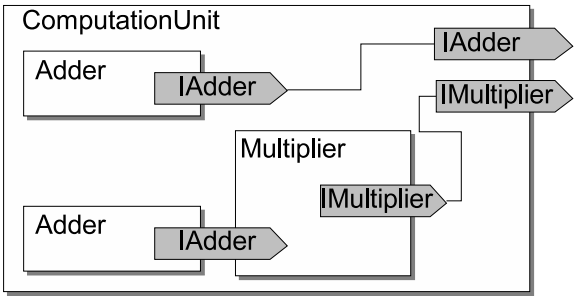

Figure 5.2: A component declaring two subcomponents of the same type

Of course, in this example, it makes no sense to declare two `Adders` instead of connecting the service of one `Adder` to both `Multiplier` and `ComputationUnit` required interfaces. Having shared services, extending our CDL, should be considered to allow synchronization or pre and post conditions to satisfy properties in a concurrent or asynchronous system. In this kind of systems, sharing of a component could be prohibitive because of some of the system properties (e.g. multiple access to a same variable). In the presented work, we have omitted this kind of extension to keep the model simple.

Again, we decided if an element should be present or not in our basic initial model depending if it obscures (or not) the model and the way of reasoning about it. In this case, adding variables in the declarations does not complicate the model but facilitates the mapping into a component implementation, each variable becomes a field of the component implementation.

5.3 Component Implementation

In this section we introduce one possible implementation for a component described in our CDL. Note that the model we are describing is implementation independent, but to give concrete examples and show how CG[3] look like, we need to refer to a concrete implementation[4].

[3]Component Generator, this concept is explained in chapter 7.

[4]Another approach such as defining a completely new language of components, which integrates architecture declaration and component implementation is naturally a valid alternative as well (most models apply this approach, for example, ArchJava [2], or Java Layers [9]).

We have chosen Java [4] as our implementation language (in fact we have chosen a syntax similar to Java for our CDL in order to facilitate the mapping between the description and the implementation of a component).

The following conventions must be followed to provide an implementation that satisfy the CDL proposed in this chapter:

- Interface definitions directly correspond to Java interfaces.
- There is a Java class per component declaration called the *component class*.
- A component class may extend any other Java class.
- A component class must implement every interface declared as provided service.
- Each variable in *requires* declaration generates a private field in the component class of the same Java type that the required interface. Those fields must be initialized through a Java constructor (which ensures that the component is bound to every of its required services after its instantiation).
- Each variable declaring a subcomponent in a *contains* clause generates a private field in the component class with the same type as the subcomponent class[5]. Initialization of those fields is done taking into account the *connects* declarations.
- A *connects* clause is straightforwardly translated to instantiate internal subcomponents, component instantiation gives to each subcomponent their corresponding required parts.
- A connection between provided services or required services as delegation mechanism) must be translated into delegation of methods calls. In case of provided services, each method call will be forwarded to the corresponding subcomponent. In case of a required service, each method call of a subcomponent will be forwarded to the field declared with the type of the required service.

5.3.1 Example

Next, we show how a component implementation in Java looks like for the declarations specified in 5.2.2, following the mapping presented in the previous section.

[5]Note that it is not an interface, but a concrete component class with its associated Java type.

```
interface IAdder {
    int add(int x, int y);
}

interface IMultiplier {
    int multiply(int x, int y);
}

class Adder implements IAdder{
    int add(int x, int y){
      return x + y;
    }
}

class Multiplier implements IMultiplier{
  private IAdder helper;

  public Multiplier(IAdder adder){
      helper = adder;
  }
  /**implemented in terms of add operations*/
  public int multiply(int x, int y){
    //assumes that (+) and (-)are unavailable for Multiplier
    if (x < 0) return multiply(helper.add(0, -x), y);
     else if (x == 1) return y;
        else
         return multiply(helper.add(x, -1),
                         helper.add(y, y));
  }
}

class ComputationUnit implements IAdder, IMultiplier{
  private  Adder adder;
  private  Multiplier  multiplier;

  //initialize internal components
  //and connect them together
  public ComputationUnit(){
     adder = new Adder();
     multiplier = new Multiplier(adder);
  }

  //delegation method for IAdder
  public int add(int x, int y){
    return adder.add(x, y);
  }

  //delegation method for IMultiplier
  public int multiply(int x, int y){
    return multiplier.multiply(x, y);
  }
}
```

In this work we have not studied the possibility of building any tool to check automatically that an implementation obeys a defined architecture, instead, we have related as much as possible the Java implementation language and our proposed CDL, taking care of not loosing generality in our model. In the future, a formalized mapping between CDL and the implementation language should be considered in order to build such a tool.

Chapter 6

Specialization Scenarios

Specialization scenarios are declared during component implementation. The specialization opportunities must be identified according to provided services of the component[1]. When a programmer identifies a specialization opportunity for a given component, she writes a specialization scenario for the interface through which a component publishes such a service. In this chapter we propose a syntactic notation for declaring the specialization opportunities.

6.1 Introduction

Software components are defined to be used in a variety of situations (they have to be able to adapt and respond according to a given context). Typically, the component producer allows the component consumer to adapt a component through a certain set of parameters. This parameterizations relies most of the times on data structures declarations and global static definitions.

In terms of provided services, these *adaptation parameters* are sometimes mixed with the rest of the service parameters. According to the usage context, a parameter will be considered as either an adaptation parameter or a normal parameter. We consider that adaptation parameters are associated with static information that does not change during the rest of the component life cycle, therefore, they represent a possible *specialization scenario*. In other words, a specialization scenarios encapsulate the adaptation opportunities offered by a component. However it is the programmer who decides whether a given static parameter is useful for specialization purposes or not, by defining a specialization scenario including this parameter.

The declaration of those specialization opportunities is done during component

[1]Actually, we will show later that the specialization may include assumptions over the required services as well.

implementation. It consists of explicitly defining what is the context of usage (that could be benefit from specialization) for a given service. Since adaptation aspects are defined in a separate declaration and not in the component implementation, they do not clutter component source code and they can be easily modified.

Explicit declaration of specialization opportunities was introduced by Le Meur et al. [31] and a formal description was presented in terms of a declaration module language: a declaration is associated to a given source program, and there is the possibility to create dependencies between modules by importing and exporting module declarations (a module requires explicitly a definition of another module). See section 3.4 for further details.

6.2 Independence of Specialization Opportunities

From our point of view of component orientation, explicit dependencies between specialization modules are not possible, because this would produce dependencies between the associated components. Let us assume a component `C` requiring a service `X` and providing a service `Y`. In our CDL we make no assumptions about which component will provide the required service `X` to `C`. Now let us suppose that during the implementation stage we identify some adaptation opportunities for the provided service `Y` and because of how `Y` is implemented we realize that required service `X` is involved. The problem is that we do not know which component will provide `X`, it could be `Z`, or `W` or any other component providing the service `X` (see figure 6.1), therefore we cannot determine which specialization declarations will be available for `X`: It is not possible to include a dependency between the specialization scenario declared by `C` and the one declared by `Z` because `Z` is not known while implementing `C`!

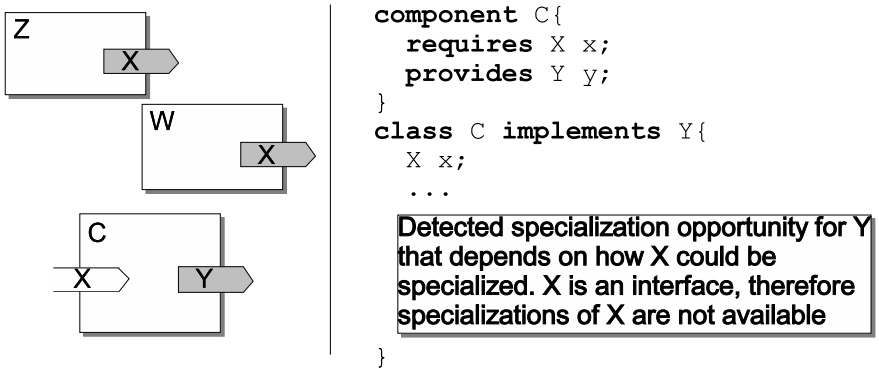

Figure 6.1: Dependencies between independently produced components

Instead of making imports and inter-module declarations we propose to declare

assumed scenarios for specializations that depend on the availability of other scenarios in required services. Because we cannot be sure that those assumed scenarios will be really available when the components have been connected, we always have to provide a safe (default) scenario that applies if such an assumed scenario[2] does not exists.

Note that, compared with figure 6.1 an entirely different situation is introduced if `C` does not requires the service `X` but contains a subcomponent Z that implements `X` (figure 6.2)

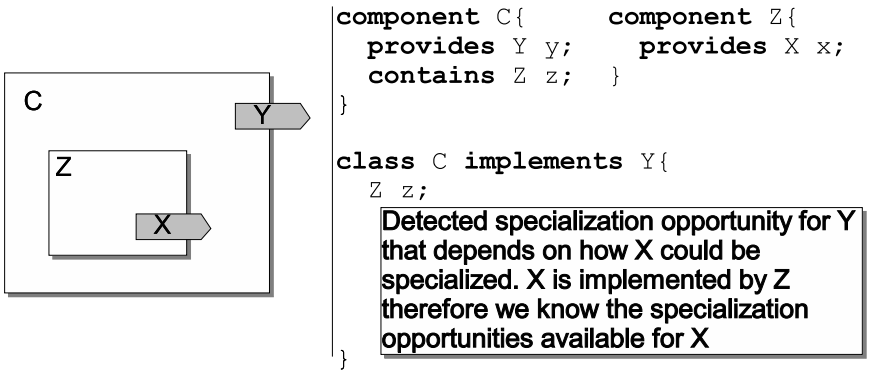

Figure 6.2: Dependencies in compound components

In this case, when implementing `C`, we will know `Z` as well, and in our definition of a specialization scenario for `Y` we could include explicit references to some specialization scenarios published by `Z` (this is the case of existing component specialization proposals such as Le Meur's).

6.3 Declaring Specialization Scenarios

A specialization scenario is bound to an interface definition rather than to the component itself. We do that just to present the concepts involved in a clear way[3].

Specialization scenarios declare the availability of fixed values[4] of service parameters that can be used to specialize the associated component implementation.

Service declarations can be seen as method signatures, and the values that can affect the binding time of the method statements (the implementation of a service) are given by the binding time of the parameters. We define a specialization scenario

[2]Note that an assumed scenario is declared as a result of some dependency between X and Y that affects how they can be specialized.

[3]With the definitions introduced in chapter 7, the reader will see that a specialization can be defined together with the component generator declaration.

[4]Static binding time, see 4.2.1.

as a tuple of binding-time values associated with the return value and each of the parameters.

A *scenario* declaration represents a possible context of execution of the method, in terms of binding time information associated with its return value (if not void) and its parameters.

A *specialization* declaration is defined over a given Java interface. A specialization declaration groups *scenario* declarations over the methods corresponding to the given Java interface. Each method signature can be associated with any number of scenarios. A specialization may include one or more *assumes* declarations which allows to make hypotheses about context information of methods in other interfaces (usually interfaces declared as required by the component). *Assumes* declarations can be seen as internal specialization declarations used by some of the scenarios to make assumptions about the availability of some scenario for a method that is not part of the same interface.

A *scenario* declaration (part of a specialization declaration) may include a clause *assumes* specifying the assumed context for a method belonging to an external interface (this assumed context must be declared in the specialization declaration using an *assumes* declaration). This declaration can be referenced inside a scenario declaration in order to make explicit that such a context information is dependent on another method.

The information about which will be the concrete components providing some services declared as required is not available during the deployment of the component (actually of its component generator). The assumptions made using *assumes* clauses and declarations are the only way to take advantage of specialization opportunities that could depend on how other components can be specialized. An *assumes* clause limits the use of a specialization scenario, imposing a strong condition: instead of only considering the binding time of the parameters, the availability of the assumed scenario in another component must be checked (see the example in section 6.3.2).

Specialization scenarios will be chosen during component specialization to determine the right binding time to be used in a service implementation. A specialization scenario is part of the input value that a binding time analyzer receives to calculate the binding time of a service implementation. Specialization scenarios allow alternative binding time annotations over the nodes that form part of a service implementation, in fact, the specializer does not work with one single annotated syntax tree but several of them (each one corresponding to different specialization scenarios).

6.3.1 Syntax

Here is the syntax to describing the elements mentioned above:

Conventions:

Program text (keywords and program separators) are in **bold**
Meta-variables are in *italic*

[x] means optional occurrence of x (zero or only one occurrence of x)

x | y means either x or y

x* means zero or more occurrences of x

x+ means one or more occurrences of x

Domains:

identifier ∈ Identifier

type ∈ Java Class Type | Java Primitive Type

bt ∈ **{S, D, void}**

Syntax:

spec_def ::= **specialization** *spec_name* **specializes** interface_name**{**
*assumes_def**
method_scenario_def+
}

assumes_def ::= **assumes** *spec_name* **specializes**interface_name**{**
single_method_ scenario_def+
}

method_ scenario_def ::= *method_signature***{**
scenario_def+
}

single_method_ scenario_def ::= *method_signature***{**
single_scenario_def+
}

method_signature ::= *type method_name parameter_list*

parameter_list ::= () |
(*not_empty_parameter_list*)

not_empty_parameter_list ::= *var_def*, *no_empty_parameter_list* |
var_def

var_def ::= *type var_name*

scenario_def ::= **scenario** *return_bt scenario_name bt_list*
[**assumes** *scenario_assumed_list*];

single_scenario_def ::= **scenario** *return_bt scenario_name bt_list*

bt_list ::= () |
(*bt_not_empty_list*)

bt_not_empty_list ::= *bt_var_def*, *bt_list* |
bt_var_def

scenario_assumed_list ::= *scenario_name* |
scenario_name, *scenario_assumed_list*

bt_var_def ::= *bt var_name*

Auxiliary definitions (only to help comprehension of part of the semantics)

spec_name ::= *identifier*

scenario_name ::= *identifier*

interface_name ::= *identifier*

method_name ::= *identifier*

var_name ::= *identifier*

return_bt ::= *bt*

We are considering only completely static or dynamic data (the binding times proposed are S, D or void). This is only a first approach, if we would like to include references as parameters or as return value (it would be the case of objects), we need to provide binding times for partially static data. Moreover, including references implies to take into account aliasing information as well.

6.3.2 Example

Following example of `Multiplier` and `Adder` components, below there are some possible scenarios definition

```
specialization AdderSpec specializes IAdder{
    int add ( int x, int y){
       scenario  S  addsc1 ( S x,  S y);
       scenario  D  addsc2 ( D x,  S y);
    }
}

specialization MultiplierSpec specializes  IMultiplier {
    assumes AssumedAdderSpec specializes IAdder{
       int add ( int x, int y){
         scenario   S  dep1add  (S x,  S y);
       }
    }
    int multiply ( int x, int y ){
       scenario   S  sc1mult (S x, S y)  assumes dep1add ;
    }
}
```

Notice that when the component producer develops the `Multiplier` component, she may not have much information about the `Adder` component. The only available information that she knows is the interface `IAdder`. This is why `MultiplierSpec` needs to define *assumed scenarios*. An *assumed scenario* allows assumptions about scenarios defined over other interfaces. Those other assumed specialization are internally declared using the clause `assumes` (instead of `specialization`). In this case we have called `AssumedAdderSpec` the assumed specialization for services declared in `IAdder`).

The example cited can be interpreted in the following way: if `dep1add` for the method `add` on `IAdder` is available then the component is able to provide the scenario `sc1mult` for the method `multiply`. Otherwise, the only scenario available is the default (which is implicit).

We are considering unique scenario names, again this is a restriction to help keep the model simple. This restriction can be easily solved including identifier containing name spaces references like Java does with types and packages.

Variable names following a binding time declaration in a scenario definition (such as `x` and `y` in `scenario S sc1mult (S x, S y)`) just appear to make the scenario more readable, they could be omitted and the order of the method parameters can be used to determine to what parameter corresponds a given binding time. For example, a shorter notation including the return value as first parameter in the list could be `scenario sc1mult = (S, S, S)`.

Chapter 7

Component Generators (CG)

In this chapter we describe the main notion of our approach for building adaptable and efficient software components. Generating extensions (described in 4.3) are extended in order to consider different specialization scenarios as part of their definition. We call *component generators*(CG) those *enhanced* generating extensions.

7.1 Introduction

Following our approach, a producer deploy component generators instead of the components. Component generators are a special kind of generating extensions, it can be made automatically using what is called a program generator generator (historically called cogen, see 4.3). This program receives the annotated abstract syntax tree (AST) of the component source code (annotated in terms of binding times)[1] and uses this information to produce a specialized component generator of the original component. The relationship between components (required and provided services and internal connections) are transformed into relationships between component generators. Instead of composing and connecting components the consumer combines component generators [7]. Once combined, a component generator can produce a specialized final component being provided the custom values (the static data).

A component generator can be seen as the factory for a family of specialized components[2]. Each of the generated components provide and require the same functionality, but optimized (specialized) to work in a different context (the specified for

[1]The input data of the component generators generator is a bit more complex that the one used in an ordinary cogen. A component generators generator has to deal with multiple annotations for each node, depending of the declared specialization opportunities, see section 8.3.

[2]It is the main difference with ordinary generating extensions, where there is only one possible way of specialize the component

the selected specialization scenario). How wide is a family depends on how many specialization opportunities were declared, and how can be combined together.

7.2 CG Philosophy

In a software component setting we have two main roles: the producer, who implements and delivers a component; and the consumer, who acquires a component (possibly from a market or component repository) and integrates it into a system to get a complete application.

It could be possible, that a consumer becomes also in producer if instead of building a final application, she builds a new component (using the acquired component as a subcomponent of the new one).

Introducing CGs implies that the product delivered by a producer (and therefore, acquired by a consumer), will not be the component itself, but its corresponding CG.

The distinction between a component and its CG is transparent at architectural level. In fact, we can see a component generator as a meta-component, in the same sense that we have classes and meta-classes in some object-oriented environments; the CGs form a parallel web with respect to the web of components. A component consumer acquires CGs instead of the components and combines them following the same architecture as the one she defined with components. With standard components, the component consumer has to configure and insert the components into the complete system. Using, CGs, the process is the same, except that is not the CG which is inserted into the system but the specialized component produced by the CG after having specified some custom values.

In order to run the specialization process, the consumer has to determine the specialization scenario (see section 7.5) according to the adaptation information that the component offers, and run the CG with the static values associated with each scenario. This process can be partially automatized providing a tool that constructs the specialization scenario declaration according to the values entered by the consumer.

Note that in the case where the consumer is at the same time a producer of a new component, the acquired CGs do not need to be executed but just combined to form a new CG. Only when a final consumer decides to use the new CG, will it be necessary to run it, in order to produce the desired specialized component (as a result of the generation of the specialization of its internal components).

CGs component specialization combines two activities:

In a first activity, there is a gathering of specialization information, the CG will automatically propagate the context information related to each specialization scenario throughout their internal service implementations and sub CGs (which will construct the internal subcomponents).

In the second activity, every CG generates a specialized component, using the collected information, which consists of decisions about what is the specialization to be applied for each service together with the static values provided. The result of this stage is a set of specialized components prepared to interact together using the more efficient version of each service in terms of the context where they will be executed.

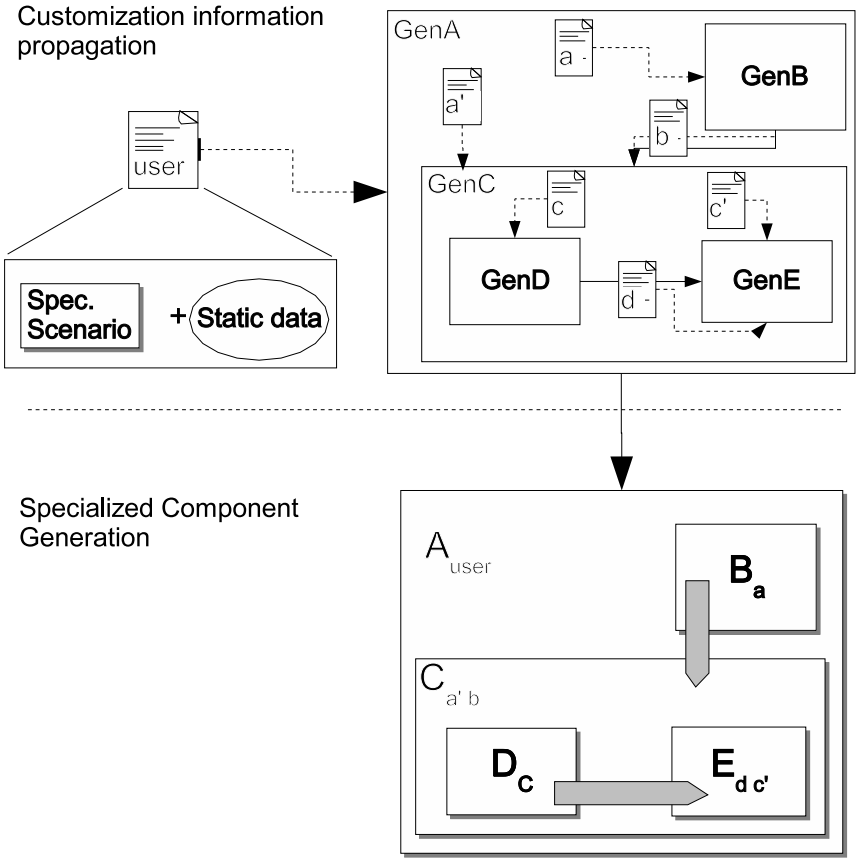

Figure 7.1: Propagation of adaptation information and component generation

In figure 7.1, we can see these two activities (presented in a sequential way although, in parctice, they could happen concurrently). User adaptation information is given to the top-level CG *GenA*. *GenA* processes the information[3] and sends the results (*a* and *a'*) to the internal CGs, *GenB* and *GenC*, which continue with the propagation process. *GenB* sends the information *b* to *GenC*, and finally *GenC* propagates the result of processing *a'* and *b* (labelled as *c* and *c'*) to *GenD* and *GenE*.

[3]In simple terms, to process the information means to create the appropriate specialization scenarios and compute required static values for interacting with some service of another CG.

After all the CGs have received the appropriate adaptation information, each of them generate the final specialized components.

7.2.1 Advantages and Drawbacks

The CG approach enables fully automatic specialization. One of the problem with fully automatic program specialization is that the user can get a program less efficient that the non-specialized version (the case of over-specialization). Or it could be that the automatic specialization does not cover all the specialization opportunities, we may obtain a better version but not the optimal one (it is called under-specialization), both problems are cited in [16]). In our proposal, specialization scenarios are defined by the component producer, who is able to analyze if a given specialization is convenient or not. However, it could be very complicated to identify the quality of the produced specialization, in particular, because of the many possible combinations of scenarios. This problem could be tackled by building a tool that presents (like a *preview*) some kind of abstract view of all the possible specialization that would be generated by the CG. The component producer can evaluate the result instead of run the CG every time she wants to measure the impact of defining some specialization scenario.

CGs preserve the architectural structure of the components that they generate. Therefore, there is no additional effort introduced by composing and linking CGs instead of components. In other words, the user (component consumer) has to compose CGs and provide adaptation information for each service in a similar way she would have to do with standard adaptable components.

In our model, component declarations and linking define a static architecture. Consequently, the CGs generate static structures of specialized components. These structures are highly tied (a consequence of eliminating indirections, offering efficient code). If a user wants to replace one component by another, she has to stop the system, replace the old CG from the linked CGs that built the previous system with the CG of the new component, and re-run the code generation for *all* components in the system. It may be an expensive and perhaps prohibitive procedure and it needs to be done carefully because it could lead to incompatibilities with information managed for previous components (for example, cache or temporary data, etc). This happens, because there is no explicit separation between the two activities described above. A way of gathering and recording useful specialization information, in order to speed up specialization if the component is replaced, could be considered. Of course, this persistence of information would not be useful in some context where the new component may introduce radical changes to the way the rest of its interacting partners can be specialized. It is necessary to study in more detail this suggestion.

7.3 CG Declaration

A CG declaration binds the definitions of specialization scenarios with component declarations. This declaration is the input for our cogen (the CG generator).

Note that the cogen needs also to receive as input each of the inner CG (which should be previously generated) for those components that are composed.

7.3.1 Syntax

This is the syntactic declaration of a CG:

Conventions:

Program text (keywords and program separators) are in **bold**
Meta-variables are in *italic*

x | y means either x or y

Domains:

identifier ∈ Identifier

Syntax:

genspec_def ::= **genspec** *genspec_name* component *component_name***{**
spec_clause
inner_genspec_clause
}

spec_clause ::= **specialization** *spec_list***;**

spec_list ::= *spec_def* |
*spec_def***,** *spec_list*

spec_def ::= *spec_name* **for** *component_variable*

inner_genspec_clause ::= **genspec** *genspec_list***;**

genspec_list::= *genspec_def* |
*genspec_def***,** *genspec_list*

genspec_def ::= *genspec_name* **for** *inner_component_variable*

Auxiliary definitions (only to help comprehension of part of the semantics)

genspec_name ::= *identifier*

spec_name ::= *identifier*

component_variable ::= *identifier*

inner_component_variable ::= *identifier*

7.3.2 Example

A declaration that specifies the specialization scenarios of a component `ComputationUnit`, and the component generator of its subcomponents (`Adder` and `Multiplier`)

```
genspec GenComputationUnit component ComputationUnit{

    /** publishes  specialization scenarios for
    *   provided services */
    specialization AdderSpec for add,
                   MultiplierSpec for mult;

    /** defines the component generator
    * (already existing ones) to be used
    *  by internal components of ComputationUnit */
     genspec GenAdder for adder,
             GenMultiplier for multiplier;

}
```

A component generator provides and requires the same services as the component being specialized. Therefore the connections between component generators is given implicitly by the connections declared in the component.

To summarize, the creation of a CG for a given component involves the following steps:

1. To write specialization scenario specifications for provided services (`AdderSpec` and `MultiplierSpec` declarations).

2. To define the generator for the component, as well as for each of the subcomponents (the `genspec` clause inside the `genspec` declaration).

7.4 Composing CGs

With this model, composing components is turned into composing CGs. How does it affect the generated specialized versions? How is the specialized version of a compound CG generated?

Each component, whether a primitive or a composed one, has an associated component generator. A component generator of a component is an object that can be instantiated being provided the corresponding instances of the component generator of each of its sub-component as well as each of its required components. The `connects` clauses used in component configuration apply also for component generators, but replacing a component type by its corresponding component generator. In this way, component generators can be composed and code generation fired in a top level component will be propagated through the sub-component generators provided during the instantiation of the top-level component generator. Consequently, the specialized components will be adapted to the current configuration (the combination of component generators).

Due to the acyclicity of the dependency graph obtained from component composition, there should be a component generator that corresponds to each primitive component. This primitive component only provides services (requiring nothing).

7.5 Interaction Between CGs

CGs interact toegether to obtain the static information needed to fulfill the declared services of the specialized component. A CG contains all the variants of a method defined according to the associated specialization scenarios. Once the service is required by another CG, it will provide the associated specialized code to the final component.

Let us consider a CG for the `Adder` component (described in section 5.2.2), it will have two variants for the method `add` according to the defined scenarios, called `sc1add` and `sc2add` (we use the name of the scenario to identify each variant of `add`). Then, when `GeAdder` is connected to another CG, let us say `GeMultiplier`, they interact in the following way: suppose a call to `multiply(int x, int y)` (where both parameters are static, and `Multiplier` has defined a scenario which can return a static result.

In this case, it means that the call should be solved when executing `GeMultiplier` component generator.

```
GeMultiplier m = new GeMultiplier(new GeAdder());

// defines a spec for multiply where
//both x and y are static s = (S, S): S
SpecializationScenario s =...
```

```
  //we suppose 3 and 4 are Objects for simplicity
  Object[] statics = {3, 4};
  //no dynamic parameters:
  String[] dynamics = {};
  m.multiply(s,dynamics,statics);
```

The body of `multiply` will dispatch the more appropriate specialization for `s`.

```
// method dispatcher of GeMultiplier :
public String multiply(SpecializationScenario s,
                       Object[]params, String dynamics){
  SpecializationScenario anotherS;
  //...finds a matching declared scenario
  if (matches(s, anotherS)){
    //method variant for anotherS:
    //suppose it matches scenario (S D):S then
    //executes method variant, obtaina specialized code
    String codebody = sc1multiply(dynamics[0], params[0]);
    //writes method code given by sc1multiply into the
    //specialized component. Simplifying:
    //e.g. "public multiplySD(int y){ return " + codebody + "}"
    String call = "multiply(" + dynamics[0] + ")";
    return call;
  }else //look for another anotherS...
  ...
}
```

We call *method variant* each of the different methods existing for specializing a method like `multiply` (there is one method variant for each specialization scenario defined, in this case, over the method `multiply`). In this case, the method `add` of `GeAdder`[4] must be called.

In this case it need to call the method `add` of `Adder`'s component given as parameter in the constructor:

```
//In some place inside the method variant sc1multiply :
//remember that this method is call if scenario (S D):D is selected
//then, first param is static and second dynamic:
private String sc1multiply(int x, String y){
 SpecializationScenario s =...
 ...
 String[] dynamics = {y};
 Object[] statics = {x};
 helperAdder.add(s, statics, dynamics);
 ...
}
```

[4]Note that the CG GeAdder is received as parameter in the constructor of the CG of Multiplier (GeMultiplier).

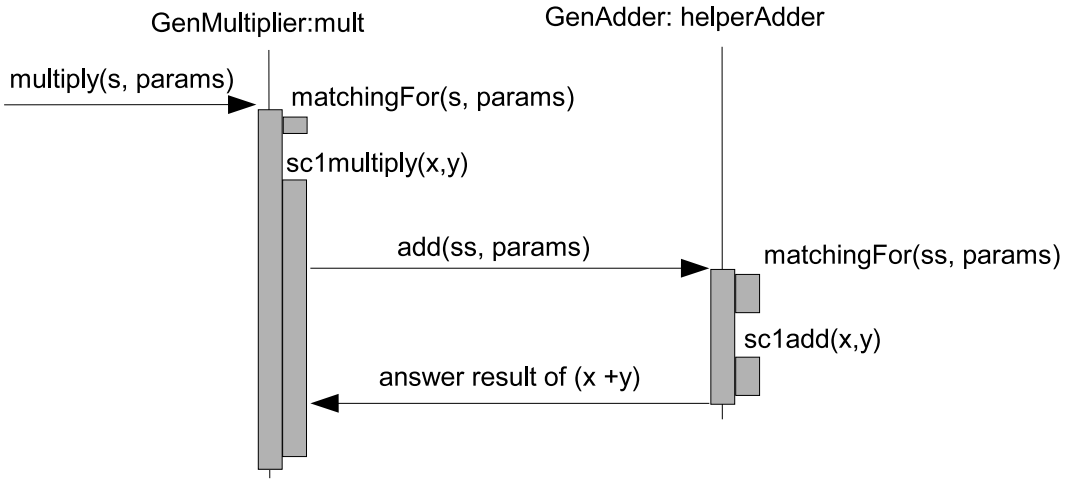

Figure 7.2: Interaction between CGs

`GeMultiplier` receives a message like `multiply(s, params)`, where `s` is a scenario declaring both `x` and `y` statics (we call it *concrete scenario*, see 7.5.2, and `params` is an array with the static values for those variables. `GeMultiplier` matches[5] `s` with some of its defined scenarios and executes the method variant associated with it, in this case, calls `sc1multiply(x, y)`. In this method variant, `GeMultiplier` needs to interact with another CG, `GeMultiplier` sends a message of the kind `helperAdder.add(ss, params)`. `GenAdder` follows the same process, by executing in this case the method variant called `sc1add` and finally returning the result of `x + y`. This is the situation described in figure 7.2.

Note that method variants are private (`sc1multiply` or `sc1add`): they are not published by the component. In fact, only the dispatcher, which executes internally one of those private variants(`multiply(s, params)` or `add(s, params)`), is known as part of the interface that allows CGs to interact.

Section 8.2.1 shows a more detailed example with code that interacts with those specializations of `Multiplier`.

[5]This "matching" is defined later, in section 7.5.2.

7.5.1 CG Interfaces

So far we have outlined what CGs are, and how they need to interact in order to propagate the static information provided to the top level CG, in order to generate the specialized components.

However, we have omitted type information in the example of `GeAdder` and `GeMultiplier`. When a component declares a required element, as happens with `Multiplier` requiring `IAdder` service, this requirement is translated into the generator `GeMultiplier` as a *field* that must contain an object (another CG) that can answer to the CG service call for the services declared in `IAdder`. Just declaring that the CG of `Adder` implements the `IAdder` interface will not work. If we come back to the example of the previous section, the message that the `Multiplier` will invoke over that field is

`helperAdder.add(sc, params, dynamics)`

This message is the result of a statement present in the `Multiplier` component implementation, and calling the service `add` defined in `IAdder`. But where is this `add(sc, params)` declared? And what is the type of `helperAdder` variable? Note that when building the CG `GeMultiplier`, the component that will provide the service `IAdder` is not know (as a consequence, its CGs are not available), therefore we do not have any known type for the variable `helperAdder`, implementing the method `add` as described.

This problem can be easily solved by defining *CG interfaces*. If we have a look at the kind of interface we need for solving such a call, we can see that every method signature in the interface will produce a call to a method with the following signature:

```
String methodName(SpecializationScenario sc,
                  Object[] statics, String dynamics)
```

That is, the return value is replaced by the type `String`, which will be the answer that the CG will include in the residual code generated,of course, instead of String it can be another object that provides more information about the returned value(is someone) and the code(if exists) to be residualized by the caller. The first parameter contains an object instance of `SpecializationScenario` (the representation of specialization scenario declarations, it is explained in next section). The array of `Object` will contain any static value that the CG must pass to the other CG[6]. The argument dynamics, contains an array of parameters dynamics (the `String` code of those parameters when the method is invoked, e.g. "x" or an expression like "x+1").

[6]By doing that we loose information about parameter types and quantity and overloaded methods would not work, however it can be solved using some renaming conventions that ensure unique method names, avoiding overloading in the CG interfaces, we have omitted this part for the sake of simplicity.

In the same way that we have an analogy between CG and component, we can have pairs of public interfaces to be used by the component (the original one, with their normal parameters and return type), and the CG (a version automatically deduced form the original interface following the given convention). Each provided service means that the CG must implement the CG interface corresponding to the type of the provided interface, and for each required service specified in a component declaration, there will be a field declared with the type of the corresponding CG interface.

In pseudo code, a CG implementation using CG interfaces can be:

```
class GeMultiplier extends GeneratingExtension
  //an implemented CG interface per provided interface
  //declared in Multiplier
  implements IGeMultiplier{
      //a field for each required service in Multiplier
      private IGeAdder helperAdder;
      ...
}
```

In this code, we have chosen to start each CG interface with IGe. Then, for the user interface `IAdder`, we have the corresponding CG interface called `IGeAdder`.

We have discussed CG interfaces here to provide a better description about how CGs work and to complete the mentioned example. The reader should notice that CG interfaces are transparent to the producer or consumer (note that they are not present in any of the syntactic languages that we have defined), they are a part of our proposed implementation for the cogen and the CGs.

7.5.2 Concrete and Declared Specialization Scenarios

We can distinguish between two kinds of scenarios with different purposes:

A *declared* specialization scenario is the one defined by the component developer (the producer), and it states the specialization opportunities for a given service (in particular, scenarios for each method). This information is provided to the cogen in order to generate the CG for a component.

A *concrete* scenario is the one that declares the usage context for a given service of a component. This information is provided to the CG to generate the specialized component. It includes the concrete values for each of the static parameters. This is the kind of scenarios that the consumer provides to the top-level CG to start the propagation of context information.

A CG will *match* the concrete scenario with some of the declared scenarios. For example, a simple rule for matching scenarios can be: given the concrete scenario A and the declared scenario B, A *matches* B if for each parameter in A, its binding time is lower than or equals to the binding time of the same parameter in B (considering that static is lower than dynamic).

The algorithm could match the first one that satisfies this condition or it can be smarter by looking up for the *matching scenario* with as many static values as possible.

In order to specialize the component, the CG will use the matching declared scenario. A necessary condition here is that a CG always has a matching scenario for any concrete scenario (it is easy to achieve this condition, it is enough to include a declared scenario which states that every attribute is dynamic).

Chapter 8

Putting the Pieces Together

In this work we have introduced the fundamental pieces for a software component development based on the creation and distribution of interacting component generators. This chapter presents a global vision of the approach, including some guidelines about the construction of a CG generator. Even though we have focused our work on the existence of CGs as component generators, there are some other parts involved in our proposal that are also discussed in this chapter.

8.1 Introduction

We have defined a component model for delivering black-box adaptable components stressing the necessity of efficient execution. The difference with other approaches relies on the degree of independence with respect to the required services that we propose. Our proposal replaces the core component implementation with a component generator, a special kind of generating extension of the component (CG, defined in chapter 7). It allows automatic component specialization without breaking component encapsulation [7]. Producers deliver to the client a CG that can be composed with other CGs in a way similar to what would happen with ordinary components.

In the following, we give an overview of the whole process, involving the building and delivery of a CG. We offer some guidelines about the construction of the CG generator and finally we describe briefly other tools and pieces that may be part of the model.

8.2 Production and Delivery of CGs

So far, we have described a component model (in chapter 5), specialization scenarios (in chapter 6), and CGs in chapter refCGs. Now, let us have a look at the whole

proposal, involving annotated component implementations, specialization scenarios and generated components and how they can be delivered and integrated into a system.

As we said before, a component consumer can become a producer as well, if she acquires a component to integrate into a new component instead of using the component to build a final application. For us, a component producer actually builds a component generator and a component consumer use this generator.

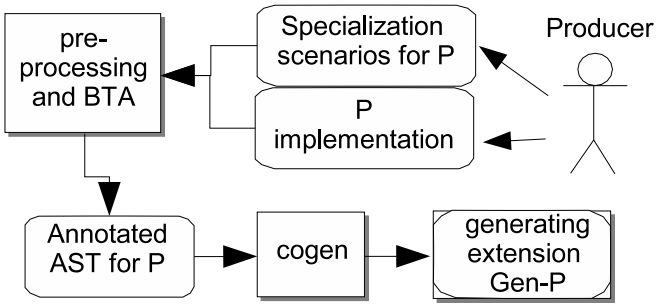

Figure 8.1: Producer's perspective: CG generation

From a producer perspective, she needs to deliver the CG of the component. In order to do that, the component implementation together with the specialization information is pre-processed (verifying the structure and the specialization declaration with the CDL specification of the component, and finally running a binding-time analysis). The result is an annotated abstract syntax tree that serves as input to the cogen, which builds the corresponding CG (figure 8.1). Note that if the component is a compound component, the internal CGs must be generated before, because they will be required by the enclosing CG.

From a consumer's point of view, the consumer acquires a CG, e.g. Gen-P, and unlike with the producer, she will not create a new CG with Gen-P but run it with some configuration values and obtain an adapted (dedicated) component that can be integrated in her system. Configuration and execution of a CG could be a very complicated task that may require connection with other CGs, however it is a process that can be automatically done having the CG specifications and information about the context provided by the user, we call *deployment unit* the module that connects, configures and runs a CG in order to obtain a final specialized component. This process is described in figure 8.2).

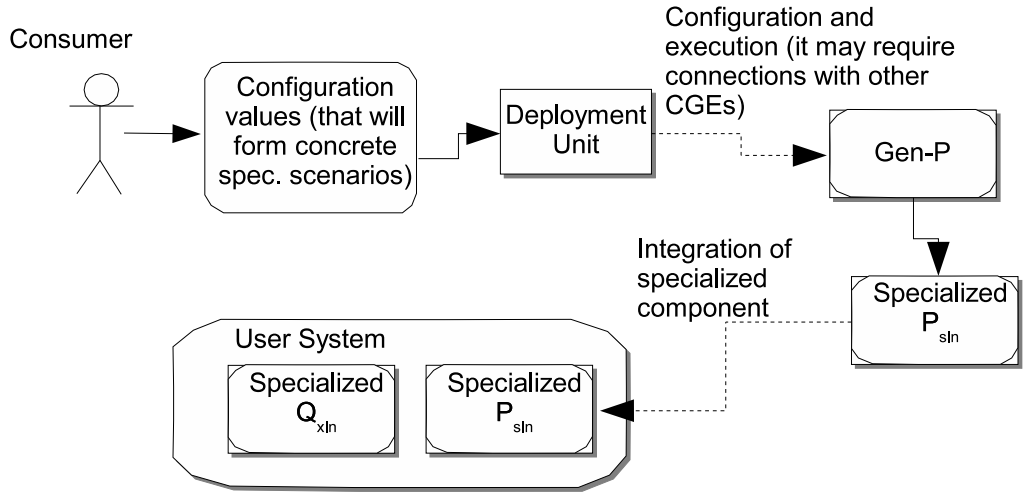

Figure 8.2: Consumer's perspective: CG execution

8.2.1 Example of CG Implementation

Let us have a look to the function power (an example cited in many places to show a simple case of applying partial evaluation, see chapter 4), in terms of our model. We recall the definition of the interface `IMultiplier` (a service provided by the component `Multiplier` in the example given in section 5.2.2) to show an interaction between CGs.

Interface and Component Declarations. Defines component architecture.

```
interface IMultiplier{
  int multiply(int x, int y);
}


interface IPow{
  int power(int n, int x);
}


component Pow {
  provides IPow power;

  requires IMultiplier mult;
}
```

Specialization Scenario Declaration. Declares a specialization opportunity for a method published in a interface.

```
/**Defines a specialization opportunity for a method in IPow
 */
specialization PowSpec on IPow{
  int power(int n, int x){
     D powern (S n, D x);
  }
}
```

CG Declaration. Binds the declared component with the specialization opportunity defined.

```
/**Associate the declared scenario with the function power
implemented by Pow */
genspec GenPow component Pow{
  specialization PowSpec for power;
}
```

Component Implementation. The implementation of the declared component.

```
class Pow implements IPow{
  IMultiplier mult;

  public Pow(IMultiplier m){
   mult = m;
  }

  int power(int n, int x){
    if (n == 1) return x;
        else return mult.multiply(x, power(n-1, x));
  }
}
```

From those specifications and the component implementation, the cogen should generate a CG implementation, where `IGenPow` is the CG interface compatible with the method power (see section 7.5.1):

```
/** method dispatcher for power
*   it returns a call to the proper specialization
*   and creates specialized method body and
*   signature in the specialized component
*/
public String power(SpecializationScenario s,
            Object[] statics, String[] dynamics){
  SpecializationScenario scen;
  //simplifying SpecializationScenario creation:
  scen = (S, D):D;
  //test each available scenario, to select
  //the appropiate specialization
  if (matches(s, scen)){
    //then:
    // specialization selected: powerSD
```

```
    // method variant to call: powern

    //retrieves static values
    Object param = statics[0]; //e.g. 4
    //obtains String code corresponding to
    //dynamic param:
    String dynparam = dynamics[0]; //e.g. "x"

    //calls the method variant powern
    //it will answer the specialized code
    String codespecialized = powern(param, dynparam);

    //generates signature of powerSD
    //i.e.: "public int powerSD(int x){"
    //generates method body of powerSD
    //using codespecialized
    //i.e.: "return " + codespecialized + ";}"
    ...
    //creates the code call to powerSD
    //(the specialized function)
    //simplifiying:
    String code = "powerSD(" + dynparam + ")";
    //answer the call, e.g. "powerSD(x)"
    return code;

    }else
   // attempts to match with next scenario
   scen = ...;
   if matches(s, scen){
     ...
}

/** method variant for spec powern:
*   n is static in specialized final component,
*   but it remains as a variable in this CG.
*   Answers the code body of the specialized function
*/
private String powern(int n, String x){
  if (n==1) return x;
   else{
    // we have to multiply x * (powern(n-1))
    // both params are dynamics
    //simplifying SpecializationScenario creation:
    SpecializationScenario s = (D, D):D;
    String[] dynamics = {x,
                         powern(n-1, x)
                        };
    //no statics for mult:
    Object[] statics = {};
    return mult.multiply(s, dynamics, statics);
    }
  }
}
```

The invocation of `multiply` returns a String with the code to be inserted into the specialized method `powerSD` (it will be an invocation to the specialized version of `multiply`, specialized by the CG of the component provided to `Pow`).

Note that the code for each function variant of the generator(in this case, just `powern(int x)`) preserves a structure quite similar to the original function (if we omit special declarations like arrays used as parameter when the method of the CG of `mult` is invoked).

Connecting this CG with the CG of the component `Multiplier` defined previously and running them with `n = 3` would lead to the next specialized component `Pow`[1]:

```
class Pow{
  Multiplier mult;

  public Pow(Multiplier m){
   mult = m;
  }

  public int powerSD(int x){
    return mult.multiply(x, mult.multiply(x, x));
  }
}
```

This is clearly a more efficient version than the original definition of `Pow` if it is reduced to the context where `n` is always 3. It eliminates recursion and applies a straight multiplication of the argument. Note that in the specialized component, the implementation of all the participants is known. The type of `mult` is not an interface but a class, therefore, the call to `multiply` can be faster than the original version, where `mult` is declared with an interface type.

Figure 8.3 sums up the main parts involved in this example. We omitted the component `Multiplier` to simplify the picture and keep the essential parts.

8.3 Building a CG Generator

We have shown how a CG looks like and how it can interact with other CGs to form a complete system. As we have explained before, if the producer had to build by hand a CG each time she wanted to deliver a new component, her work would be very tedious, painful, error prone and would require much more time. The solution is to have a tool that automatically generates a CG for a given component, the cogen that we have described in 4.3.

Following direct approach for generating a cogen [18], we can mention the following

[1] In this case the result of the interaction leads to a call to the default `multiply` method. There is no scenario defined for arguments with binding time `(D S)` in the component `Multiplier`.

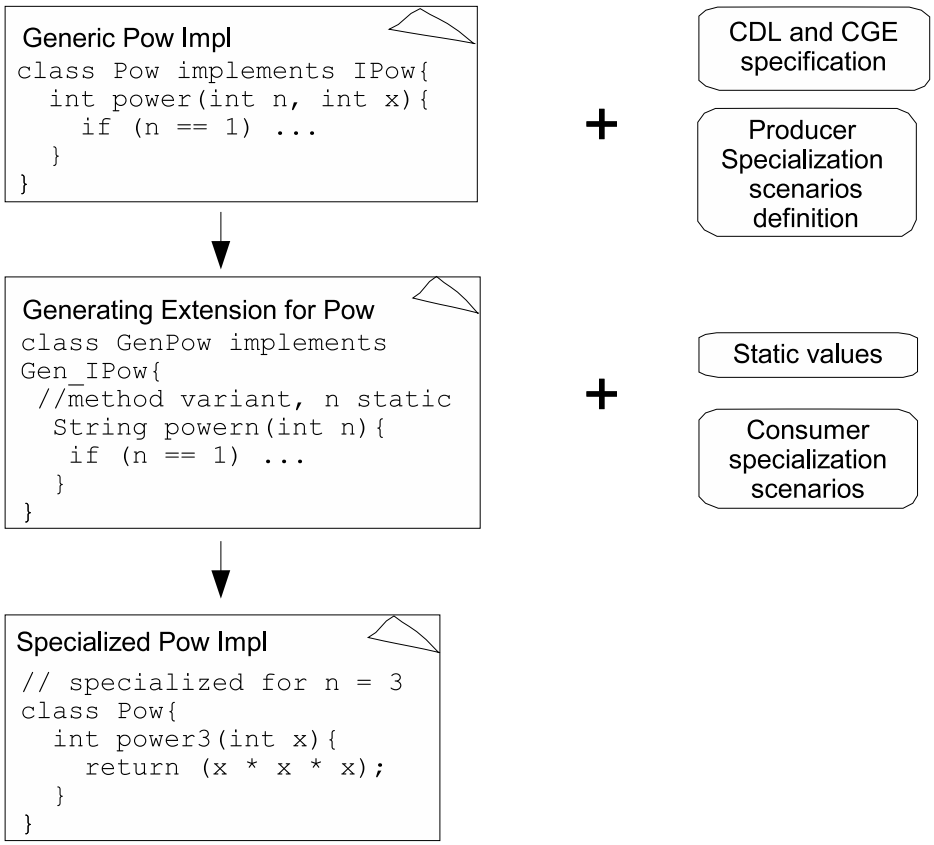

Figure 8.3: CG for the `Pow` component

requirements in terms of inputs and outputs that can serve as a guide for building such a tool.

Each component is either a primitive component or is composed of other components. In the first case, the cogen will receive only the implementation of the component and the component generator specification as input, but in the second case, the cogen needs the other already produced CGs as well.

Source code corresponding to component implementation will be processed by the cogen, and consequently it does not need to be delivered to the consumer, only the resulting compiled CG will be delivered.

The CG code must not be very much larger than the component itself. This can be achieved using a component structure similar to the component implementation. In fact, the only considerable extra code should correspond to the different variants for a service defined in specialization scenarios (specialization scenarios avoid code

explosion by limiting specialization only to explicitly declared opportunities: the cogen does not need to consider any possible variant during CG building).

The main input of a cogen is the annotated source code of a component implementation. This source code can be handled by means of its corresponding abstract syntax tree (AST). The nodes of this tree are augmented with binding time information, indicating the static or dynamic nature of each subtree. Note a different annotated AST is required for each specialization scenario. Instead, what we can do is to append, to each node, a binding-time annotation for each specialization scenario.

Given a function (a part of the offered services), there will be as many specialization variants as scenarios were defined for it (in the example described in section 8.2.1, `powern` is one method variant, another one could be `powerx` for a scenario that declares `x` as static and `n` dynamic).

The building of the component generator for a given function and specialization scenario consists of producing code to *residualize* or *rebuild* the appropriate parts of the tree as program tex while computing the static parts of the tree. *Residualize* consists basically of remaining unchanged the static computations (they will remain in the code of the CG) and *rebuild* implies generate code that reconstructs the original code.

The direct approach for building such a cogen can be simpler because it only does syntax manipulation of input source code (traversing and rebuilding of syntax trees).

The hard work here is done previously to running cogen, in the example given above (in section 8.2.1), we do not really give the source of `Power` to the cogen, instead we have to build the annotated version (in terms of binding times, and the given specialization scenarios) of its AST first. However, how to obtain such an annotated version in an automatic way is a topic by itself and we will not get into details about binding-time analyzers for object-oriented environments.

8.4 Verification, Analysis and Deployment

This work is mainly focused on the creation of a CG generator, however, our proposal needs some other pieces in order to produce the appropriate input for the CG as well as for the cogen itself. In the following, we discuss briefly these other parts of our model.

8.4.1 CDL Verification and Code Generation

We have defined CDL (see section 5.2) as a declarative way of describing a component architecture. Given a formal description of how the concepts expressed with CDL should be mapped in the component implementation language (in our case, we have defined informally a mapping to Java code), it is possible to build a verifier that checks

validity of the implementation with respect to a given CDL declaration. A further step would be to generate the implementation skeleton for the target language. In this way, the user only fills service method implementations.

On the other hand, it could be a valid alternative to have a complete component language, instead of separating component declarations (CDL) from implementation (in this case Java) and providing a mapping between them, it is possible to join both languages in one single component language. The code written in such a language can be automatically translated into Java or C code to be able to apply existing binding-time analysis or code optimizer, before running the cogen to obtain the corresponding CG.

8.4.2 Binding-Time Analyzer

The quality of a specialized component will depend on the appropriate declaration of specialization opportunity but also, fundamentally on the correctness and precision of the binding-time analysis that produces the annotated abstract syntax tree taken as input by the cogen. In our prototype, we have done this analysis manually, but to go further, a binding-time analyzer should be provided.

Efficient binding time analysis for object-oriented languages continues is an open research area. Ther is, for instance, some work in progress using binding-time analysis based on constraints (BTA analyzer for petitCafé [39]).

8.4.3 Deployment Unit

As explained before, CG generation and execution is a process that involves component producers as well as component consumers (see section 8.2). At some point in the process of deploying and acquiring of CGs, there will be a final consumer that may want to get the specialized versions generated by the CG (the reason of its existence). This can be done by a configuration script that instantiates and properly connects the involved CGs and finally runs the top-level CG with the specialization scenarios and values defined by the user. This configuration script can be automatically made, by what we have called a deployment unit. Such a tool can read the CG specifications and the user defined specializations and, following conventional rules,it can run the specialization process to instantiate and to connect the CGs.

8.5 The Proposed Model

Combining the elements presented in the previous chapters and the new ones defined in this chapter, we obtain the architecture described in figure 8.4.

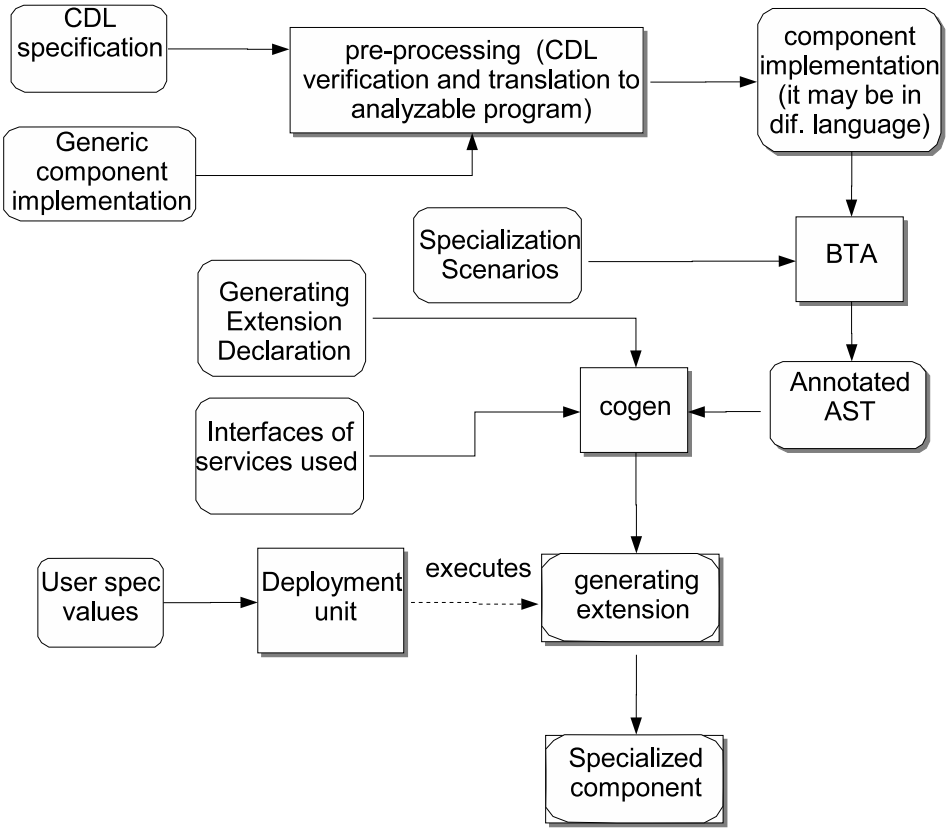

Figure 8.4: Software component specialization model

The rounded rectangle means user input information (which can be producer or consumer data according to the destination process). Shadowed rectangles show processing units, like cogen, BTA unit and deployment unit. Shadowed rounded rectangles indicate outputs of a processing unit, CG and specialized components are represented by rounded rectangle inside a normal shadowed rectangle because they are output of other processes and at the same time processing unit.

8.6 The CG Generator Prototype

We have made a prototypical version of part of the model presented above. The implementation covers the code generator generator (cogen) and it is based on a component model implemented following the guidelines suggested in 5.3 about how

to get a possible implementation of the given CDL in Java.

In order to interact with it and to generate a CG from a component, we have used the Eclipse platform [26] to build a *plugin* that, by starting from a simple component implementation (corresponding to just one Java class), allows the user to create specialization scenarios for each of the methods, and permits to mark parts of code that must be considered static. In this way, we simulate the steps that should be done through a binding-time analyzer.

Figure 8.5 shows how the plugin looks like when the user open a component class and add some specialization scenarios and binding-time information into the code. In this case, the user has defined a specialization scenario called `dsd (S D):D`, which assigns static binding time to the parameter `x` and dynamic binding time to the parameter `y`. Static binding times are shown in the code underlined with a squiggly line.

Having defined some specialization scenarios and selected the static parts for each of them, the user can activate the CG generation process. It produces a new file with the Java source code corresponding to the CG generated. With some limitations[2], and only considering simple expressions and basic data types, the CG generated is compilable and executable, generating specialized components according to the provided context.

Figure 8.6, shows part of the code of the CG corresponding to the example component shown in the figure 8.5. The field containing a subcomponent generated a field in the CG containing another CG (corresponding to the component `AnotherCompo`). For each method in the component there is a dispatcher with the same name as the original method (`mparam`), but including specialization scenario information as parameter. This method will be called by other CGs that need to interact with this one. When a method dispatcher is called, it returns a `String` containing the code that the caller should insert in its specialized component. This code will be a static value, like an integer, or in case of the `mparam`, where the return value is dynamic, it will return a method call of the form `scenarioName-mparam(3, var)`. A method dispatcher will also call the appropriate *method variant*, one variant of `mparam` per defined specialization scenario. In the figure one can see part of the method variant called `dsd-mparam(int x)` according to the specialization scenario received as parameter and the available scenarios defined in the CG. The purpose of a method variant is to execute the static parts of the original method body (for example a computation involving the static value of `x`) and to generate the residual code into the specialized component.

[2]Strong checking of proper context information propagation throughout the sub CGs, other steps like checking of component composition (proper composition and compatibility checks), and detecting cycles in the collaboration graph must be considered.

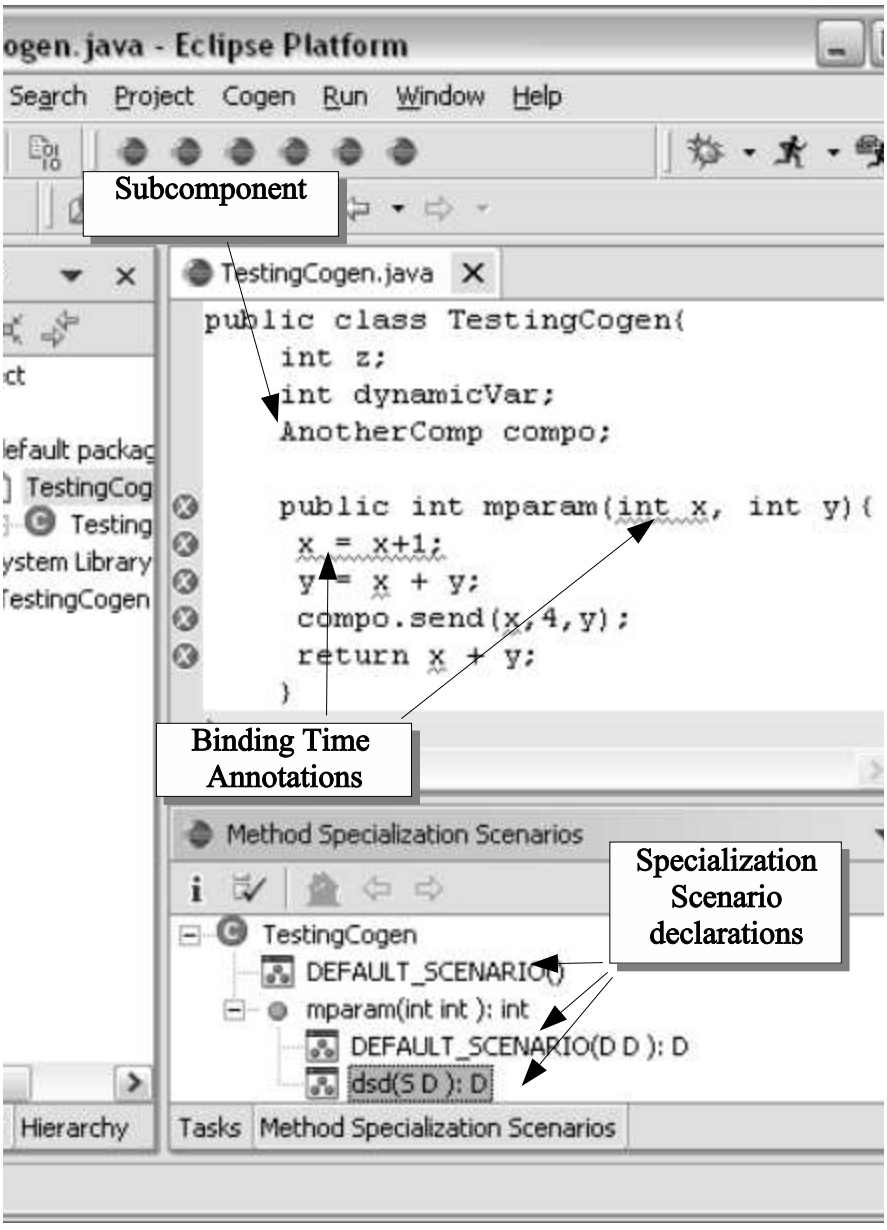

Figure 8.5: The CG generator plugin

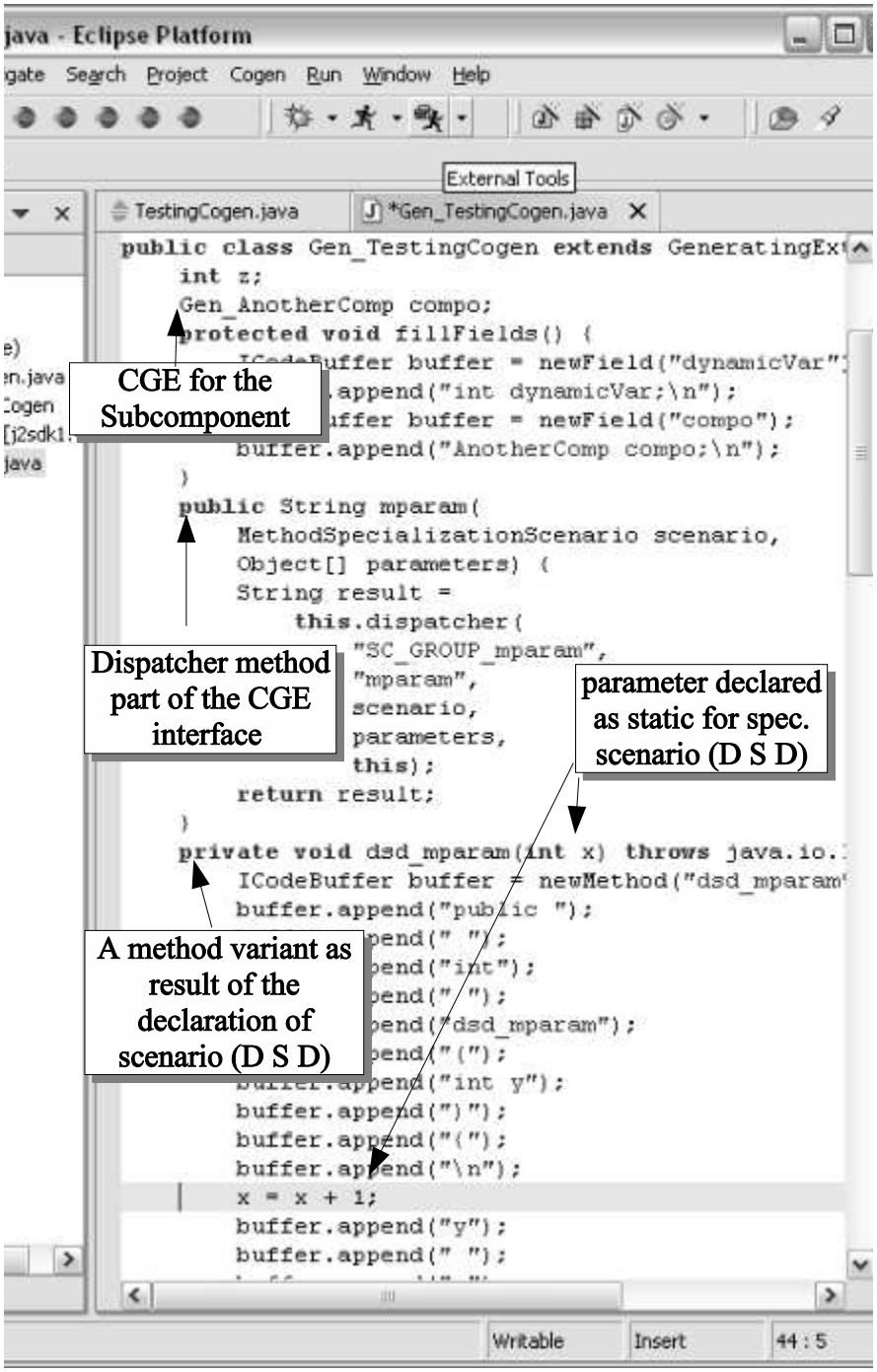

Figure 8.6: The generated CG

Chapter 9

Conclusion

This chapter reviews the concepts and objectives covered by this work. We present some related projects and we offer some perspectives about future work.

9.1 Related Work

Our idea of providing an explicit language (CDL) to separate component architecture from implementation is not new, in fact it appears in a family of predecessors of modern software component concepts, the Module Interconnection Languages (MIL) [40]. The main purpose of MILs is to help software engineers during design, evolution and maintenance of the system, allowing automatic checking of system structure. The CDL follows the same goals, it can be used to automatically verify the component implementation, checking composition and binding between provided and required services, guaranteeing the correct generation of the corresponding CG. However, while MILs provide a description and verification of the global structure of a software system[1], a CDL specifies only a part of that system, the component being described. The CDL allows to declare the required services of a component without connecting them to other components.

Our aim of obtaining automatic specialization of software component is inspired by the authors Le Meur and Consel in [31][2]. They suggested to use explicit declaration of specialization opportunities during component implementation. Specialization opportunities are grouped in a "module" associated to a specific "component" implementation. The process consists of delivering a specializable component (consisting of

[1]Latest MILs considered separate compilation of modules, however, the interconnection between the modules remains static. Components declare functionalities as required, but they do not specify which component will provide such functionalities.

[2]A proposal about working with program specialization using a black-box model was suggested previously by Schultz in [43].

the implementation and the declared specialization). Afterwards, when a consumer acquires this specializable component she has to use a linker that puts the components together and applies the specialization process. The user specifies the custom values (adaptation information) to the linker, obtaining as a result a component specialized for the user's application. The process works well when the user wants to build a top-level component that does not depend on any other component (i.e., any required service is satisfied)[3], but it falls short of satisfying the needs of intermediate component producers (those who want to deliver a new component integrating existing ones, eventually leaving some required and provided services unbound). The proposal of Le Meur and Consel is based on Tempo, a specializer for C code [12].

Schultz et al. extended partial evaluation to object-oriented languages, in particular for the Java language. The result is a specializer for object-oriented programs called JSpec [27], based on Tempo[4]. JSpec uses specialization classes in order to provide some form of modular specialization [43]. However, there are no precedents with respect to building component generators (in terms of generating extensions) for object oriented languages.

We have put together, specialization scenarios with the idea of self specializable components, using the notion of generating extensions and hand-writing program generator generators, a concept covered by Heldal and Hughes [25, 18, 24] between others.

9.2 Future Work

The scope of this work is only preliminary. The implications and feasibility of a realistic implementation still require a deeper study.

Future work implies the study of a concrete implementation of our component generator generator. It includes the creation of the tools suggested in chapter 8.

An important issue to be dealt with in an immediate future is the formalization and implementation of a proper binding-time analysis, it is the only way to obtain reliable annotated programs that really can be used to test our model. In order to do so, it may be necessary to restrict the Java language to a simpler subset that enables an easier reasoning about desirable properties. One subset candidate could be petitCafé[39], over which a constraint-based binding time analysis is already defined. This analysis can be improved and adapted without inconveniences to our proposal.

Finally, a study of the benefits of the proposal over a real case (or at least a good approximation to it) is necessary to verify the capabilities of the whole model.

[3] We have called *final consumer* this kind of user.

[4] Actually, it uses Tempo run-time specialization facilities to at run time generate binary programs

9.3 Conclusions

We have presented the problem raised when attempting to delivery efficient and adaptable software components. However, it is not straightforward to capitalize upon these opportunities while keeping a pure component model delivering independent and adaptable black-box components.

We claim that specialization techniques, widely applied to traditional software systems composed basically of modules or libraries, need to be extended and adapted to work properly with software components.

Potential specialization opportunities are captured in our proposal by allowing component producer to declare specialization scenarios following the suggestion of Le Meur[31].

Focusing our work on the notion of generating extensions[21, 29] and Birkedal's proposal of hand-writing program generator generators[6], we have presented a model that consists of deployment of component generators (a special kind of generating extensions), programs that are able to create specialized versions of a component driven by the static information available at consumption/assembly time and specialization scenarios declared at production time.

We have offered some guidelines about how to automatically generate component generators. To demonstrate the feasibility of our proposal we have built a prototype, stressing simplicity in order to facilitate reasoning about it and to make implementation tractable in the short period of time available.

Bibliography

[1] *Corba components*, OMG TC Document orbos/99-02-05, March 1999, Joint Revised Submission. 15

[2] J. Aldrich, C. Chambers, and D. Notkin, *ArchJava: Connecting software architecture to implementation*, Proceedings of the 24th International Conference on Software Engineering (Orlando, FL, USA), IEEE Computer Society Press, May 2002. 11, 33, 39

[3] L.O. Andersen, *Program analysis and specialization for the C programming language*, Ph.D. thesis, DIKU, University of Copenhagen, May 1994. 22

[4] K. Arnold and J. Gosling, *The Java programming language*, 2nd ed., Addison-Wesley, 1997. 3, 34, 40

[5] S. Baker, *CORBA distributed objects using Orbix*, Addison-Wesley, 1997. 12

[6] L. Birkedal and M. Welinder, *Hand-writing program generator generators*, Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP '94) (M. Hermenegildo and J. Penjam, eds.), Springer-Verlag, 1994, pp. 198–214. 2, 30, 79

[7] G. Bobeff and J. Noyé, *Component specialization: Towards deeper adaptation*, Systèmes à composants adaptables et extensibles (Grenoble, France), October 2002. 2, 51, 63

[8] M.A. Bulyonkov, *Extracting polyvariant binding time analysis from polyvariant specializer*, Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation - PEPM'93 (Copenhagen, Denmark), ACM, June 1993. 2

[9] R. Cardone and C. Lin, *Comparing frameworks and layered refinement*, Proceedings of the 23rd International Conference on Software Engineering (Toronto, Canada), IEEE Computer Society Press, May 2001, pp. 285–294. 39

[10] C. Consel, *A tour of Schism: A partial evaluation system for higher-order applicative languages*, Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation - PEPM'93 (Copenhagen, Denmark), ACM Press, June 1993, pp. 66–77. 26

[11] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and E.N. Volanschi, *Partial evaluation for software engineering*, ACM Computing Surveys **30** (1998), no. 3. 1

[12] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and E. N. Volanschi, *Tempo: Specializing systems applications and beyond*, ACM Computing Surveys **30** (1998), no. 3. 78

[13] C. Consel, L. Hornof, F. Noël, J. Noyé, and E. N. Volanschi, *A uniform approach for compile-time and run-time specialization*, Partial Evaluation, International Seminar, Dagstuhl Castle (O. Danvy, R. Glück, and P. Thiemann, eds.), Lecture Notes in Computer Science, vol. 1110, Springer-Verlag, February 1996. 22, 26

[14] C. Consel and S.C. Khoo, *Parameterized partial evaluation*, Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (Toronto, Ontario, Canada), ACM, ACM Press, June 1991, ACM SIGPLAN Notices, 26(6), pp. 92–106. 1

[15] C. Consel and F. Noël, *A general approach for run-time specialization and its application to C*, Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA), ACM Press, January 1996. 22

[16] J. Dean, C. Chambers, and D. Grove, *Selective specialization for object-oriented languages*, Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, ACM Press, June 1995, ACM SIGPLAN Notices, 30(6). 54

[17] L.G. DeMichiel, L.Ü. Yalçinalp, and S. Krishnan, *Enterprise JavaBeans^{TM} specification*, SUN Microsystems, August 2001, Version 2.0, Final Release. 15

[18] D. Dussart, R. Heldal, and J. Hughes, *Module-sensitive program specialization*, Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (Las Vega, NV, USA), ACM SIGPLAN Notices, 32(5), May 1997, pp. 206–214. 30, 68, 78

[19] D. Dussart, F. Henglein, and C. Mossin, *Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time*, Proceedings of

the Second International Symposium on Static Analysis, SAS'95 (Glasgow, UK) (A. Mycroft, ed.), Lecture Notes in Computer Science, vol. 983, Springer-Verlag, September 1995, pp. 118–135. 29

[20] R. Glück and J. Jørgensen, *Efficient multi-level generating extensions for program specialization*, Proceedings of the 7^{th} International Symposium on Programming Language Implementation and Logic Programming (Utrecht, The Netherlands) (M. Hermenegildo and S. Doaitse Swierstra, eds.), Lecture Notes in Computer Science, no. 982, September 1995, pp. 259–278. 2

[21] R. Glück and J. Jørgensen, *Efficient multi-level generating extension for program specialization*, Proceedings of the 7 International Symposium on Programming Language Implementation and Logic Programming (Utrecht, The Netherlands), no. 982, september 1995, pp. 259–278. 2, 22, 23, 79

[22] Object Management Group, *CORBA components*, Adopted Specification formal/02-06-65, OMG, June 2002, Version 3.0. 33

[23] R. Guerraoui (ed.), *Ecoop'99 - object-oriented programming - 13th european conference*, Lecture Notes in Computer Science, vol. 1648, Lisbon, Portugal, Springer-Verlag, June 1999. 84

[24] R. Heldal and J. Hughes, *Partial evaluation and separate compilation*, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (Amsterdam, The Netherlands), ACM Press, June 1997, ACM SIGPLAN Notices, 32(12), pp. 1–11. 78

[25] R. Heldal and J. Launchbury, *Handwriting cogen to avoid problems with static typing*, Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming (Skye, Scotland), Glasgow University, 1991, pp. 210–218. 2, 29, 78

[26] IBM, *http://www.eclipse.org*. 3, 73

[27] INRIA/Compose, *http://compose.labri.fr/prototypes/jspec*. 78

[28] N.D. Jones, *An introduction to partial evaluation*, ACM Computing Surveys **28** (1996), no. 3, 480–503. 1

[29] N.D. Jones, C.K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*, International Series in Computer Science, Prentice Hall, 1993. 23, 25, 26, 27, 29, 79

[30] J. Jørgensen and M. Leuschel, *Efficiently generating efficient generating extensions in Prolog*, Report CW 221, Department of Computing Science, Katholieke Universiteit Leuven, February 1996. 2

[31] A.F. Le Meur, C. Consel, and B. Escrig, *An environment for building customizable software components*, IFIP/ACM Working Conference - Component Deployment (Berlin, Germany), Springer-Verlag, 2002, pp. 1–14. 22, 44, 77, 79

[32] J. Lloyd and J. Shepherdson, *Partial evaluation in logic programming*, Journal of Logic Programming **11** (1991), 217–242. 26

[33] V. Matena and M. Hapner, *Enterprise JavaBeans^{TM}*, SUN Microsystems, March 1998, Version 1.0. 12, 20

[34] S. McDirmid, M. Flatt, and W.C. Hsieh, *Jiazzi: New-age components for old-fashioned Java*, OOPSLA'01, Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, 2001. 14

[35] R. Monson-Haefel, *Enterprise JavaBeans*, 2nd ed., O'Reilly, March 2000. 33

[36] F. Nielson, H.R. Nielson, and C. Hankin, *Principles of program analysis*, Springer-Verlag, 1999. 28

[37] D. Parnas, *On the criteria for decomposing systems into modules*, Communications of the ACM **15** (1972), no. 12, 1053–1058. 13

[38] D. Parnas, *Designing software for ease of extension and contraction*, IEEE Transactions on Computers (1979), 128–138. 14

[39] L. Ponisio, *A binding-time analysis for petitCafé*, Master's thesis, Vrije Universiteit Brussel and École des Mines de Nantes, August 2002. 71, 78

[40] R. Prieto-Diaz and J. M. Neighbors, *Module Interconnection Languages*, The Journal of Systems and Software **6** (1986), 307–334. 77

[41] T. Reps and T. Turnidge, *Program specialization via program slicing*, Partial Evaluation, International Seminar, Dagstuhl Castle (O. Danvy, R. Glück, and P. Thiemann, eds.), Lecture Notes in Computer Science, vol. 1110, Springer-Verlag, February 1996, pp. 409–429. 1, 22, 25, 26

[42] U.P. Schultz, *Black-box program specialization*, in Guerraoui [23]. 22

[43] U.P. Schultz, *Object-Oriented Software Engineering Using Partial Evaluation*, Ph.D. thesis, Université de Rennes I, December 2000. 1, 26, 77, 78

[44] U.P. Schultz, J. Lawall, C. Consel, and G. Muller, *Towards automatic specialization of Java programs*, in Guerraoui [23], pp. 367–390. 22

[45] SUN Microsystems, *JavaBeans^{TM}*, July 1997, Version 1.01. 20

[46] C. Szyperski, *Independently extensible systems- software engineering potential and challenges*, Proceedings, 19th Australasian Computer Science Conference, Australian Computer Science Communication, 1996, pp. 203–212. 15

[47] C. Szyperski, *Component Software*, 2nd ed., Addison-Wesley, 2002. 12

[48] F. Tip, *A survey of program slicing techniques*, Tech. Report CS-R9438, CWI, 1994. 1, 26

[49] T.L. Veldhuizen, *C++ templates as partial evaluation*, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (San Antonio, TX, USA), ACM Press, January 1999, pp. 13–18. 26

[50] A. Weinand, E. Gamma, and R. Marty, *An object-oriented application framework in C++*, Special Issue of SIGPLAN Notices, 1988. 8