Increasing Reuse in Component Models through Genericity

Julien Bigot[1] and Christian Pérez[2]

[1] LIP/INSA Rennes
[2] LIP/INRIA
{julien.bigot, christian.perez}@inria.fr

Abstract. A current limitation to component reusability is that component models target to describe a *deployed* assembly and thus bind the behavior of a component to the data-types it manipulates. This paper studies the feasibility of supporting genericity within component models, including component and port types. The proposed approach works by extending the meta-model of an existing component model. It is applied to the SCA component model; a working prototype shows its feasibility.

1 Introduction

Component based software engineering is a very interesting approach to increase code reusability. Component models are used in a variety of domains such as embedded systems (Fractal [1]), distributed computing (CCM [2], SCA [3]), and even high performance computing (CCA [4]).

There is however usually a direct mapping between component instances and execution resources as well as between components and the data-types they manipulate. This means that a component implementation binds together three distinct concerns: the behavior of the component, the data-types it manipulates and the execution resources it is targeted to. Separating those three concerns would greatly increase reusability as each aspect could be selected independently to be combined latter. While there are some works on automatic mapping of components on resources, there are few works on abstracting component models similarly as what is addressed by generic programming where algorithms and data-types can be parameters.

This paper presents an attempt to support genericity in component models in order to validate the feasibility of this idea and evaluate its advantages on an example. This is achieved by extending an existing component model (SCA) with concepts to support genericity and by implementing a tool that transforms applications written in this extended model back to the original one.

The remainder of this paper is organized as follow. Section 2 describes an example of component that would benefit from the introduction of genericity. Section 3 analyzes some related works. An approach to introduce genericity is described in Section 4 and applied to SCA in Section 5. Then, Section 6 evaluates how this generic SCA can be used to implement the example described earlier through a prototype. Finally, Section 7 concludes and present some future works.

2 A motivating example: the task farm

Algorithmic skeletons are constructs that describe the structure of recurring composition patterns [5]. Some skeletons have been identified for the case of parallel computing, such as the pipeline (computation in stages), the task farm (embarrassingly parallel computations), the map and reduce (data parallel apply to all and sum up computations), the loop (determinate and indeterminate iterative computations) and the divide & conquer skeletons.

As an example, the task farm skeleton shown in Fig. 1 takes a data-stream as input and outputs a processed version of this stream. The parallelism is obtained by running in parallel multiple instances of workers (`W` in Fig. 1), each one processing a single piece of data at a time. A dispatcher (`D` in Fig. 1) handles the input and chooses the worker for each piece of data and a collector (`C` in Fig. 1) reorders the outputs of the workers to generate the farm output.

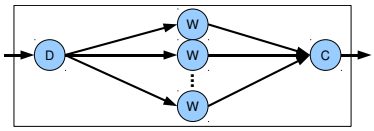

Fig. 1: The task farm skeleton.

A typical component based implementation of the task farm will wrap each of the three roles in a component. The farm itself will be a composite containing instances of these components. To increase the reusability of this composite, it should be possible to use it for various processing applied to the data, for various types of data in the stream and for various numbers of workers. It is thus interesting to let the type of the data stream, the implementation of the workers as well as their number be parameters of the composite.

When knowledge about the content of the manipulated data or about the behavior of the workers makes it possible to provide optimized implementations of the dispatcher or collector it should be possible to use these implementations. This should however not complexify the usage for cases where the default implementation is sufficient. It is thus interesting to let the implementations of these two components be parameter of the composite with some default values.

Hence, the task farm is a good example of component that could benefit from the support of genericity. Kinds of needed parameters include data-values (the number of workers), data-type (data stream) and component implementations (dispatcher, worker and collector), with the possibility to provide default values.

3 Related works

3.1 Languages with support for genericity

Genericity [6] is ubiquitous in object-oriented languages. For example, ADA, C++, C♯, Eiffel and Java all support it [7]. Classes, methods and in some cases procedures can accept parameters. Parameters can be data-types or in some cases data-value constants. A typical usage is to implement type-safe containers where the type of the contained data is a parameter.

There are two main approaches for handling the validity of parameter values. In some languages such as JAVA, explicit constraints on the values of parameters [8] restrict their uses in the implementation. In other languages such as C++, the uses of the parameters in the implementation restrict the values they can be bound to [9]. Explicit constraints eases the writing and debugging of applications as invalid use of generic concepts can be detected using their public interface only. Describing the minimal constraints on parameters can however prove to be a very complex task. The upcoming C++0x [10] takes a mixed approach: constraints are expressed as use patterns of the parameters (this is called "concepts" in the C++0x terminology) but this does not prevent use of parameters in the implementation that were not covered by a "concept".

In some languages such as C++, explicit specializations can be provided for specific values of the parameter. This makes it possible to provide optimized implementations for these cases. This also makes the language Turing complete and enables template meta-programming [11].

As far as we know, there is no component model with support for genericity (except for HOCs further discussed in Sec. 3.3). The closest features found in most models are configuration properties which are values that can be set to configure the behaviour of components. In some models such as CCM these configuration properties can be modified at run-time. In other models such as SCA, they can only be set in the assembly making them more similar to generic parameters. Unlike generic parameters however, properties are only used to carry data-values, not types.

3.2 Algorithmic skeletons

As seen in the previous section, the implementation of algorithmic skeletons is an example where genericity brings great advantages. Model bringing together components and skeletons have already been described, for example in [12]. These models are very similar to a component model supporting genericity from the point of view of a user of skeletons: skeletons are instanciated and the implementation of the component it contains are passed as parameters. In these model however, skeletons are supported by keywords of the assembly language and their implementation is generated by a dedicated compiler. From the point of view of the developer of skeletons this means that supporting new skeletons or new implementations of existing skeletons requires modifications of this compiler, which can be difficult and strongly limits reusability.

3.3 Higher order components

Higher Order Components (HOCs) [13] is a project based on the Globus grid middleware. With HOCs, a Globus service implementation S can accept string parameters identifying other service implementations. At run-time, S can create instances of these services and use them, thus addressing the issue of reusable assembly structure. However, type consistency can not be statically checked as instantiation and use of services are deeply hidden in S implementation. Another

limitation is that only service types can be passed as parameters; data-types can not. For the task farm implementation, it leads to a distinct implementation for each data-type processed in the stream.

4 An approach to introduce genericity in component models

4.1 Overview

Introducing genericity in a component model means making some of its concepts generic. A generic concept accepts parameters and defines a family of concepts: its specializations. Each combination of parameter values of the generic concept defines one specialization.

Supporting generic concepts means that when one is used, the values of its parameters must be retrieved and the correct specialization must be used. This can either be done at run-time (as has been done for C♯ for example) or through a compilation phase (as has been done for C++ for example).

The compilation approach has the advantage of requiring no modification of the run-time. It can also lead to a more efficient result since the computation of the specializations to use has already been done. On the other hand, this approach makes it impossible to dynamically instantiate specializations that were not statically used in the initial assembly.

This paper studies the compilation approach: it describes a transformation that takes as input a set of generic components and that generates its non generic equivalent. The transformation is based on Model Driven Engineering (MDE). It manipulates two distinct component meta-models: B, a basic (*i.e.* non generic) component meta-model and $G(B)$, the corresponding generic component meta-model. The proposed algorithm to transform instances of $G(B)$ into semantically equivalent instances of B is presented in Section 4.3. The next section describes a pattern to derive a meta-model $G(B)$ from a basic component model whose meta-model is B.

4.2 Genericity pattern

As a first step, the concepts of B that will be given as parameters in $G(B)$ and those that will accept parameters (be generic) must be chosen. An example of application of the pattern described here is shown in Fig. 2. As $G(B)$ is an extension of B, all the elements of B belong to $G(B)$; this section describes the additions made to the meta-model to support genericity.

For each concept that can be given as a parameter (e.g. `PortType` in Fig. 2), a meta-class with a "name" attribute is created to model such a parameters (e.g. `PortTypeParameter` in Fig. 2). For each concept that is turned generic (e.g. `ComponentType` in Fig. 2), attributes (lines in the figure) are added to its meta-class to model its parameters.

For each concept that can be given as a parameter, an additional meta-class that references a parameter and inherits from the initial concept is created (e.g.

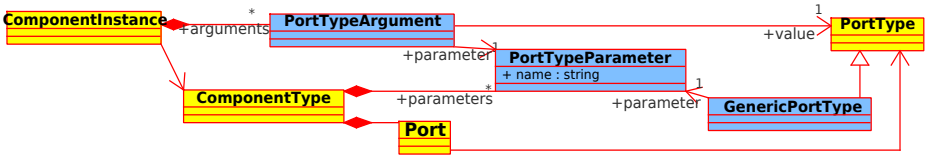

Fig. 2. Example of modifications to make `ComponentType` generic and to let `PortType` be given as parameter.

`GenericPortType` in Fig. 2). This meta-class can now be used wherever the concept given as parameter is used (e.g. `Port` references a `PortType` in Fig. 2)

For each concept that can be given as a parameter, an argument meta-class is created to reference both the parameter and its value (e.g. `PortTypeParameter` in Fig. 2). Each meta-class that references the concept made generic (`Port` in Fig. 2) has an argument attribute added.

These are the minimal additions to $G(B)$ required to support genericity. Other additions not shown in Fig. 2 can however be interesting. To support default value for parameters, an attribute referencing the value must be added to parameter meta-classes. For example, the `PortTypeParameter` will get a `PortType` attribute.

To support constraints on parameters values, two approaches can be used: either adding a constraint attribute to the parameter meta-classes or adding it directly to the generic meta-classes. As applying constraints to the parameter meta-classes prevents the expression of constraints that depend on more than one parameter, the second approach is chosen. A root meta-class for constraints must be added. The kind of constraints that can be expressed depend on the kind of parameters. For example for a data constant parameter, a range of values can be an interesting constraint while for an object interface, the interfaces it extends can be constrained. For each kind of constraint, a meta-class that inherits from the root meta-class must be added. In addition, meta-classes modeling the various logic combinations of other constraints must be added.

To support explicit specializations, a specialization meta-class must be added for each generic concept. This meta-class has a constraint attribute that specifies in what case it must be used. It also has a copy of all attributes modeling the implementation of the generic concept. The meta-class modeling the generic concept on the other hand must have a specialization attribute added that models its explicit specializations. For example, the generic `ComponentType` meta-class will contain a `ComponentTypeSpecialization` attribute. This meta-class will contain a `Constraint` attribute as well as the content of the `ComponentType`: ports, implementation, etc.

4.3 Transformation from $G(B)$ to B

This section describes an algorithm that transforms an application described in a generic component model into its equivalent in the basic component model.

The transformation algorithm takes an instance i of $G(B)$ (a set of meta-object of $G(B)$) as input and computes a semantically equivalent instance of B. The algorithm relies on a recursive function that takes a meta-object o of $G(B)$ and a context c (bindings between generic parameters and their values) as input and returns a meta-object of B semantically equivalent to o.

The main function of the algorithm iterates through all components of i. If a component can be instantiated in an empty context (at the root of an application), the recursive function is used to generate its equivalent in B. This equivalent is added to the output meta-model instance.

The recursive function generates the equivalent of o using one of the four following behaviors depending on the kind of concept the meta-class of o models.

If the modeled concept is a **generic concept** (such as `ComponentType` in Fig. 2), c is filled with the default values for parameters that have not been previously bound to a value. Then, the constraint on parameter values is checked: if it is not fulfilled, the transformation is aborted with an error. The constraints of each explicit specialization are checked. If one is fulfilled, the result of the function applied to each meta-object contained by this specialization is added to the result. Otherwise, the function is applied to the default content.

If the modeled concept **references a parameter** (such as `GenericPortType` in Fig. 2), the value of this parameter is looked up in c and the application of the function to this value is returned. If there is no binding for this parameter in c, the transformation is aborted with an error.

If the modeled concept **references a generic concept** (such as `ComponentInstance` in Fig. 2), a new context is created and filled with the arguments contained by o. Then, the result of the function applied to each meta-object contained by o with this new context is added to the result.

If the modeled concept **does not belong to any of the previous categories**, the result of the function applied to each meta-object contained by o is added to the result.

An instance of $G(B)$ is valid if it conforms to the meta-model and leads to a valid instance of B when the algorithm is applied. An example of instance that conforms to the meta-model but is invalid is a composite containing (possibly transitively) an instance of itself as it may lead to infinite recursion. The recursion can be broken if the composite accepts parameters that are used in the constraints of an explicit specialization. As genericity with recursion and selection of explicit specializations is very likely to be Turing-complete, the termination problem is expected to be undecidable. C++ compilers facing the same problem fix a limit to the recursion depth after which an error is emitted.

5 Case Study: Turning SCA Generic

5.1 Overview of SCA

SCA [3] (*Service Component Architecture*) is a component model specification. It aims at easing service oriented applications development by making possible

their description as components assemblies. It defines two types of ports: services and references, both typed by an interface. Interfaces can be extracted from various descriptors such as a JAVA interface, a WSDL interface, etc. SCA also supports configuration properties as part of the external interface of SCA components. Components can have two kinds of implementations: composite implementations provided by an assembly or native implementations (such as JAVA or C++ classes).

5.2 A meta-model for generic SCA

The pattern described in Section 4.2 has been applied to the SCA meta-model in order to create a meta-model for generic SCA. The SCA meta-model described as part of the "eclipse SCA Tools project[3]" has been used for this purpose. The concepts made generic are composites and native components. Support for generic JAVA classes has also been added. Composites accept implementations, interfaces, data-types and data-values as parameters. Native components accept data-types and data-values as parameters. Finally, JAVA classes accept JAVA types as parameters.

This required the addition of height additional meta-classes: `GenericImplementation`, `ImplementationParameter`, `ImplementationArgument`, `GenericInterface`, `InterfaceParameter`, `InterfaceArgument`, `JavaTypeParameter` and `JavaTypeArgument`. All the parameter meta-classes support default parameter value. No `GenericJavaType` meta-class has been created as JAVA types are simply identified by a string containing their name. No modifications have been done to support data-value parameters as SCA already has the concept of configuration properties.

Support for constraints on parameter values of composites and native components has been added with a root meta-class for constraints: `Constraint`. As configuration properties are referenced by xpath expression in SCA XML documents, a constraint meta-class that supports boolean xpath expressions has been added: `XpathConstraint`. The constraints supported on other kinds of parameters are currently limited to exact equality constraints supported by the meta-classes `ImplementationEqConstraint`, `InterfaceEqConstraint` and `JavaTypeEqConstraint`. Three constraints that support logical combinations of other constraints have also been added:`ConjonctionConstraint`, `DisjunctionConstraint` and `NegationConstraint`.

Support for explicit specialization of composite has been added with the addition of a `compositeSpecialization` meta-class which duplicates the content of the `Composite` meta-class.

5.3 Implementation

A prototype implementation of a generic SCA to plain SCA transformation engine has been developed. It implements the algorithm described in Section 4.3.

[3] `http://www.eclipse.org/stp/sca/`

As a special case, support for generic JAVA classes simply consists in checking JAVA type parameters for compatibility and erasing them. This is due to the fact that JAVA handles generics by type erasure: type parameters are used at compile-time for checking validity and then removed from the generated class file.

The meta-models of SCA and generic SCA are written in the ecore modeling language, A first implementation attempt has been made with a Domain Specific Language (DSL) for model transformations: operational Query View Transform (QVT). The support for this language for the transformation of ecore meta-models is however not satisfying yet and the algorithm has finally been coded in plain JAVA.

JAVA classes corresponding to the ecore meta-classes of generic SCA have been automatically generated. Those provided as part of the eclipse SCA Tools project and corresponding to the meta-classes of plain SCA have been used. The code used to instantiate these classes by parsing generic SCA XML files and to dump them in plain SCA XML file is also automatically generated thanks to annotations in the meta-model. More than 50.000 lines of JAVA code have been generated; the same amount from the eclipse SCA Tools project are reused.

The implementation of the transformation algorithm requires around 750 lines of JAVA. Most of them simply copies attributes from classes modeling concepts of generic SCA to the attribute with the same name in classes modeling plain SCA (last case of the algorithm). Those could also have been automatically generated if QVT had been used. The real logic of the algorithm only requires around 100 lines of JAVA; this is however only an estimation as it is mixed with the attribute copy part.

6 Generic Task Farm Component in Generic SCA

This sections examines the definition and implementation of a generic task farm in generic SCA. It aims at showing the feasibility of the approach.

6.1 Generic `Farm` Component

The `Farm` composite implements the task farm and accepts six parameters. Two JAVA type parameters: `I` and `O` define the type of the input and output of the farm respectively. There are three implementation parameters `D`, `W` and `C` that define the types of the dispatcher, workers and collector respectively; and an integer parameter `N` that defines the number of workers.

Its implementation is shown in Fig. 3. It simply instanciates the `D` and `C` components and relies on the `Replication` composite further described in the next subsection to instantiate multiple instance of `W`. These instances are connected by data streams simulated with a generic JAVA interface `DataPush<T>` with a single asynchronous method `void push(T data)`. This interface is used with `I` as argument before the workers and with `O` after.

The `D` and `C` parameters have default values provided: `RRDispatcher<T>` and `SimpleCollector<T>` that dispatch the data using a round-robbin algorithm

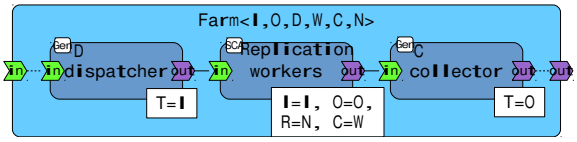

Fig. 3. The `Farm` composite

and collect them with no reordering. These are generics JAVA implementations that do not depend on the data type manipulated.

6.2 Generic `Replication` Component

The Replication composite implements the replication of a given component and accepts four parameters. Two JAVA type parameters (`I` and `O`) define the type of its input and output. An implementation parameter (`C`) defines the type of the replicated component. An integer parameter (`R`) defines the number of replications.

Its implementation shown in Fig. 4 relies on meta-programming and recursion. It contains one instance of `C` called "additional" and one instance of `Replication` with the value of `R` decreased by one. The base case of the recursion is provided by an explicit specialization used when the value of `R` reaches one.

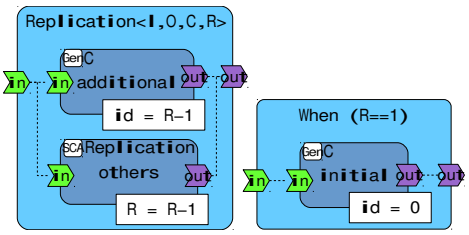

Fig. 4: The `Replication` composite.

In the non specialized implementation of the composite, the "in" service promotes two services. This is not allowed by the SCA specification. As a workaround, a concept of multiple service has been added to generic SCA that can only be connected to a reference with multiplicity "0..n" or "1..n". At transformation phase, instances of multiple services are replaced by multiple instances of classical services.

6.3 Evaluation

This implementation of the task farm has been used to compute pictures of the mandelbrot set. Two kinds of workers have been written and used with the generic task farm: one that computes the value of a single pixel at a time and another that computes whole tiles. Each version has been used in the farm with one, two and four workers. The transformation phase takes between one and two seconds and most of this time is spent parsing the input files. The resulting component have been succefully run using tuscany-java-1.4 on muticore hosts. The meta-model for generic SCA, the compiler and the source code for these components can be found at `http://graal.ens-lyon.fr/~jbigot/genericSCA`.

7 Conclusion

This paper has studied the feasibility of increasing reusability in component models thanks to genericity. To make use of existing models, the selected approach was to derive a generic meta-model from an existing one, and to provide an algorithm to transform generic component applications into non-generic ones. This has been applied to SCA and validated with an image rendering application based on a generic task farm component.

Future works include the application of this approach to others models than SCA, the comparison of the implementation of skeletons using genericity with classical skeletons, the support for dynamic instantiation of generic components and the study of the possibility to automatically compute some parameters (for example when they are targeted at specific kind of execution resources).

References

1. Bruneton, E., Coupaye, T., Stefani, J.B.: The Fractal Component Model, version 2.0.3 draft. The ObjectWeb Consortium. (Feb. 2004)
2. Object Management Group: Common Object Request Broker Architecture Specification, Version 3.1, Part 3: CORBA Component Model. (Jan. 2008)
3. Open Service Oriented Architecture: SCA Service Component Architecture: Assembly Model Specification Version 1.00. (Mar. 2007)
4. Allan, B.A., et al.: A Component Architecture for High-Performance Scientific Computing. International Journal of High Performance Computing Applications **20**(2) (2006) 163–202
5. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**(3) (2004) 389–406
6. Musser, D.R., Stepanov, A.A.: Generic Programming. In: ISAAC '88: Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation, London, UK, Springer-Verlag (1989) 13–25
7. Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J.G., Willcock, J.: A comparative study of language support for generic programming. In: OOPSLA, New York, NY, USA, ACM (2003) 115–134
8. Bracha, G.: Generics in the Java Programming Language (Jul. 2004)
9. Stroustrup, B.: The C++ Programming Language. 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
10. Gregor, D., Järvi, J., Siek, J.G., Stroustrup, B., Reis, G.D., Lumsdaine, A.: Concepts: linguistic support for generic programming in C++. In: OOPSLA. (2006) 291–310
11. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional (2004)
12. Aldinucci, M., Bouziane, H., Danelutto, M., Pérez, C.: Towards Software Component Assembly Language Enhanced with Workflows and Skeletons. In: Joint Workshop on Component-Based High Performance Computing and Component-Based Software Engineering and Software Architecture (CBHPC/CompFrame 2008). (Oct. 2008)
13. Gorlatch, S., Dünnweber, J.: From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In: Future Generation Grids, Springer Verlag (2005)