Adaptable Generic Programming with Required Type Specifications and Package Templates

Eyvind W. Axelsen Stein Krogdahl
Department of Informatics, University of Oslo, Norway
{eyvinda, steinkr}@ifi.uio.no

Abstract

The aim of this work is to provide better support for adaption and refinement of generic code. This type of flexibility is desirable in order to fully reap the potential of generic programming. Our proposal for an improved mechanism is an extension to the previously published *Package Templates* (PT) mechanism, which is designed for development of reusable modules that can be adapted to their specific purpose when used in a program. The PT mechanism relies on compile-time specialization, and supports separate type checking and type-safe composition of modules. The extension to PT presented here is called *required types*, and can be seen as an enhanced form of type parameters, allowing them the same flexibility as other elements of the PT mechanism. We implement a subset of the *Boost Graph Library* in order to exemplify, validate, and compare our approach to other options.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software; D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages

General Terms Languages, Design

Keywords Generic Programming, Reuse, Templates

1. Introduction

When developing libraries or other software components meant for widespread reuse, it is vital to minimize assumptions on the client code. On the other hand, it is equally important to be able to express a sufficient set of requirements for the clients of the library so that it can be written in a type-safe manner that yields efficient code. Furthermore, it is important that a client can refine and adapt the library to the problem at hand.

Many languages, such as e.g. Java, C#, C++, Scala, and Haskell, support constructs for *generic programming* in order to better facilitate the development of reusable libraries. The degree to which each language supports such constructs varies, and an excellent overview that compares several languages with respect to support for generic programming can be found in [11].

There are several definitions of what generic programming actually entails (or should entail), but perhaps a more fruitful angle is to consider what it is that we are trying to achieve with such mechanisms. In that respect, Jazayeri et. al [15, page 2] state the following:

> "The goal of generic programming is to express algorithms and data structures in a *broadly adaptable*, interoperable form that allows their direct use in software construction" [emphasis ours].

We agree, to a large extent, with this quote, even if it may be deemed a bit wide in scope. The adaptability part of the goal is in our opinion very important, and in this paper we will describe a mechanism that we think in many cases can represent both an improvement and a simplification with respect to adaptable generic programming compared to contemporary approaches.

The mechanism is an extension of the Package Template (PT) mechanism [3, 16]. We will in the following refer to the previously published variant as *basic PT*, or just PT when the variant is obvious from the context. Basic PT allows type safe renaming, merging, and refinement in the form of static additions and overrides that are orthogonal to ordinary inheritance. It thus differs from typical virtual class-based [17] mechanisms in that composition and refinement (beyond ordinary OO constructs) is reified at compile-time only, yielding a simpler type system.

Seeking to also attain the goal presented above for *generic*, potentially *heavily parameterized*, libraries, we

AOSD'12, March 25-30, 2012, Potsdam, Germany.

incorporate a notion of *required type specifications* in the PT mechanism, and we label this variant PT*r*. The approach is inspired by suggestions to use virtual types as an alternative approach to generic parameterization in Java [27], and enables utilization of basic PT's inherent capabilities for adaption also for generic concepts and constraints. PT*r* supports multi-type concepts, associated types, and nominal and structural generic bounds. Retroactive modeling and adaption through renaming, merging and additions are thus also supported, without sacrificing type safety, performance, or dynamic dispatch.

To demonstrate and validate our approach, we have implemented a small yet non-trivial subset of the Boost Graph Library [25], which employs a rather advanced usage of generics. The subset is the same as that described and implemented by [11], and we will compare and contrast the implementation made possible with *PTr* with those of [11]. The implementation and a prototype compiler can be downloaded from `http://swat.project.ifi.uio.no/software`.

The main contribution of this paper is thus to present PT*r* as an approach to creating flexible generic libraries, and to demonstrate through a non-trivial example its benefits and tradeoffs.

The rest of this paper is organized as follows: Section 2 presents necessary background material and introduces basic PT (2.1), and the Generic Graph Library (2.2) through a discussion on criteria for generic constructs in general and the goals of our mechanism in particular. Sections 3 and 4 contain a description of the proposed addition of required types to PT, and an overview of how PT*r* fulfills most of the criteria presented in Section 2.2, respectively. Related work is treated in Section 5, and Section 6 concludes this paper.

2. Background

2.1 A Brief Overview of the Basic PT Mechanism

In this section we give a general overview of the basic PT mechanism. The concepts of the mechanism are not in themselves tied to any particular object-oriented language, but the examples will be presented in a Java-like syntax. The interested reader is referred to [3, 16] for a more thorough exposition.

A package template looks much like a regular Java package, but we will use a syntax where curly braces enclose the contents of both templates and regular packages, e.g.:

```
template T<R> { // R is not discussed here, see Sec. 3
  class A { ... }
  class B extends A { ... } }
```

In contrast to for instance templates in C++, package templates can be type checked independently of their potential usage(s).

A template is instantiated at compile time with an `inst` statement. Such an instantiation will create a local copy of the template classes, potentially with specified modifications, within the instantiating package or template. An example of this is shown below:

```
package P {
  inst T with A => C, B => D;
  class C adds { ... }
  class D adds { ... } // D extends C since B extends A
}
```

Here, a unique instance of the contents of the package template `T` will be created and imported into the package `P`. In its simplest form, the `inst` statement just names the template to be instantiated, e.g. "`inst T`". However, modifications can also be made to the template classes upon instantiation, such as:

- Elements of the template may be renamed. This is done in the `with`-clause of the `inst`-statement, and is only shown for class names above (`A` is renamed to `C` and `B` is renamed to `D`). For renaming of class attributes another arrow is used (`->`). Note that all renaming in PT is done based on the name bindings from the semantic analysis.
- In each instantiation the classes in the template may be given additions: fields and methods may be added and virtual methods may be overridden. This is done in `adds`-clauses as shown for `C` and `D`.

An important property of PT is that everything in the instantiated template that was typed with classes from this template (`A` and `B`) is updated to instead refer to the corresponding names of the addition classes (`C` and `D`) at the time of instantiation. Any sub/supertype relations within the template are preserved in the package where it is instantiated. Note that templates can also be instantiated in other templates.

Classes from different template instantiations may be *merged* to form one new class. Syntactically, merging is obtained by renaming classes from two or more template instantiations to the same name, and they thereby end up as one class. The new class gets all the attributes of the instantiated classes, together with the attributes of the common addition class. Consider the simple example below:

```
template T { class A { int i; A m1(A a) { ... } } }
template U {
  abstract class B { int j; abstract B m2(B b); }
}
```

Consider now the following usage of these templates:

```
inst T with A => MergeAB;
inst U with B => MergeAB;
class MergeAB adds {
   int k;
   MergeAB m2(MergeAB ab) { return ab.m1(this); }
}
```

These instantiations result in a class `MergeAB`, that contains the integer variables `i`, `j` and `k`, and the methods `m1` and `m2`. Note how the abstract method `m2` from `B` is implemented in the `adds` clause, and furthermore how both `m1` and `m2` now have signatures of the form `MergeAB` $\rightarrow$ `MergeAB`.

Merging classes in this manner might obviously lead to name clashes; such conflicts must be resolved through renaming.

2.2 The Generic Graph Library and Evaluation of Generic Support

For the purpose of demonstrating and validating the generic programming constructs added to PT in this paper, we have implemented a small yet non-trivial subset of the Boost Graph Library (BGL) [25], revolving around a set of algorithms using variants of breadth-first search, including Prim's minimum spanning tree, Dijkstra's shortest paths, Johnson's shortest paths, and Bellman & Ford's shortest paths algorithms. The implemented subset is the same as that of [11].

In the implementations from [11], emphasis is put on expressing minimal requirements for each algorithm. These have internal (acyclic) dependencies, e.g. Johnson's algorithm depends on Dijkstra's algorithm.

The graph itself is represented in terms of *concepts*. The term concept is in [11] used to mean a set of requirements consisting of required operations (methods) and data type constraints. A type (or a set of types) is said to *model* a concept if it (they) fulfill(s) these requirements. In a Java-like language, concepts are typically realized as interfaces[1], and classes implementing such an interface thus model the corresponding concept. The following main concepts of the graph library are in Java implemented as interfaces:

- `VertexListGraph`: provides an iterator yielding all vertices in the graph in an unspecified order.
- `EdgeListGraph`: provides an iterator yielding all edges in the graph in an unspecified order.
- `IncidenceGraph`: provides an iterator yielding the directed edges going out of a given vertex.

These concepts are used by the algorithms to express constraints on their input parameters. For convenience, concepts that are combinations of the aforementioned ones are introduced, e.g. the `VertexListAndIncidenceGraph` concept which is an interface that extends both the interfaces representing the `VertexListGraph` concept and the `IncidenceGraph` concept. Additionally, the different algorithms require various data structures for coloring, ordering, etc., realized as e.g. property maps, queues, etc. These structures are supplied explicitly as parameters to each algorithm.

Several languages were studied in [11], and subsequently evaluated based on their support for generic programming constructs. Table 1 shows an overview of the rating for C++ and Java from that paper, plus an additional column for Scala, the latter taken from [22]. The different categories in the table are described in Table 2A, taken from [11]. An extended evaluation, including PT*r*, will be presented in Section 4.

While the original study included several other languages as well, we focus on Java and C++, and in addition we include Scala. C++ is interesting because its generic capabilities rely heavily on its templating mechanism, and it is in this language that the Boost Graph Library has its native implementation. However, C++ templates differ drastically from the templates of PT, most notably in the sense that PT templates are declarationally complete semantic units that can be type checked independently of their usage. As can be seen from the table, while much can be achieved in C++ due to its flexibility with regards to template definition, we only get limited compiler support.[2] Java is obviously interesting because the lacking points for Java in the table is the situation that we wish to ameliorate with PT*r*, and PT is designed as an extension to Java-like languages. Scala is a rather advanced JVM language that also addresses many of the weaknesses of Java with regards to generic programming, and as such it is an interesting language with which to compare and contrast our mechanism.

A partial goal for this paper can thus be summarized as *to bring some of the flexibility of C++ generic template programming to Java with PTr, while retaining compiler support, static safety, and relative simplicity*.

However, if we look back to the goal from [15] presented in the introduction, we argue that the criteria from [11] do not adequately encompass requirements for writing algorithms and data structures that are *broadly adaptable*. For instance: How can a constraint for a generic parameter be adapted to match existing code? How can a constraint be refined by subsequent users? How can a concept be adapted to allow modeling by existing data structures, or vice versa?

In order to cover these usage scenarios, we introduce two new criteria related to adaptability, presented in Table 2B. Thus, another part of our goal with this paper is *to satisfy the adaptability criteria for generic programming*.

Scores for Scala. In [22], Oliviera et al. discuss the criteria from Table 2A in context of the Scala language.

[1] See [22] for an alternative approach based on the Concept pattern.

[2] Note, however, that Java-style generics can be emulated in C++, with compiler support, through use of for instance the Boost Library's `BOOST_STATIC_ASSERT`.

	C++	Java	Scala
Multi-type concepts	*	○	●
Multiple constraints	*	●	●
Associated type access	●	◐	●
Constraints on assoc. types	*	◐	●
Retroactive modeling	*	○	●
Type aliases	●	○	●
Separate compilation	○	●	●
Implicit argument deduction	●	●	●

Table 1. *The table shows the level of support for generic programming constructs for C++, Java, and Scala. The table criteria and the support levels for the former two are taken directly from [11, page 147]. A black circle indicates full support, a half-filled circle indicates partial support and a white circle indicates poor or no support at all. For C++, a rating of '*' means that the feature is not explicitly supported by the language, but the permissiveness of the language allows one to program as if it were supported, though sans compiler support.*

In their treatment, Scala receives the *full support* verdict on all points, as shown in Table 1. One could thus think that there is little room for making improvements with PT*r*. However, we have found that, at least with respect to an implementation of the generic graph library, Scala still leaves a few things to be desired with respect to these criteria. Furthermore, as we shall see, PT*r* takes quite a different approach to generics compared to Scala. Also note that [22] changes the scores for Java, but we have kept them in their original form from [11]. We refer to the discussion of the individual criteria in Section 4 for details.

3. Required Type Specifications in PT*r*

Basic PT [3, 16] allows templates to have generic type parameters in much the same way as ordinary Java classes can, e.g. as in

```
template T<R> { ... }
```

The parameters may be constrained, either through a nominal inheritance specification (akin to Java generics) or through a structural requirements specification, e.g.:

```
template T1<R extends Runnable> { ... }
template T2<R extends { void run(); }> { ... }
```

While this provides the ability to let the template classes be collectively parameterized, which in itself can be very useful, the approach has certain limitations. To begin with, the parameter `R` in the examples above does not naturally lend itself to the kind of modification that are allowed for basic PT classes, e.g. renaming of attributes, merging etc. Since the parameter is part of the template specification, it could be natural (or even necessary) to adapt the parameter specification along with other adaptions of the template. Furthermore, neither the name `R` nor the requirement it poses is *propagated* to other templates that instantiate `T`, `T1`, or `T2`. As we will see examples of below, and as was demonstrated in [11], this is a real issue for the implementation of larger libraries as it often leads to unnecessary code duplication. Also, as a consequence of the lack of propagation, there is no straightforward way to refine constraints without cumbersome and error-prone repetition of code.

With basic PT, class declarations in a package template can, as we have seen in Section 2.1, be adapted in several ways. It seems like a natural step forward to provide the same degree of flexibility for parameterization of templates. Thus, this can be seen as making the generic constraints of a template *first class entities* of the template, in the same way that classes and interfaces are. In the rest of this section, we will look at how PT*r* provides this feature.

Required types as first-class template declarations. The syntax for required types in PT*r* is summarized in Figure 1. A basic specification requiring an unconstrained type, i.e. any Java class or interface, to be supplied can be expressed as follows:

```
template T1 { required type R { } }
```

An actual parameter for `R` can be supplied to `T1` when instantiating the template, by utilizing the `<=` arrow. To e.g. supply `String` for `R` in `T1`, the following instantiation could be used (within another template or package, see separate paragraph on that below):

```
inst T1 with R <= String;
```

At this point we see an important difference between required types in PT*r* and type parameters as found in e.g. Java, Scala or C#with regards to their scope: Required types are at the same lexical level as class declarations, and can thus naturally constrain a *set* of classes. This is to some extent similar to abstract types and nested classes within an outer class in e.g. Scala, but it is important to note that required types in PT*r* (and package templates as a whole) is a compile-time construct only. Thus, after instantiation of the template in a package, there will be no inner abstract types (and thus no full family polymorphism [8] nor path-dependent types). This amounts to a simpler type system, and is as such comparable to the flattening property of traits [23].

A required type `R` can be constrained by both structural and nominal specifications; below we see an example of the former:

```
template T2 { required type R { void run(); } }
```

Given an instantiation that supplies an actual type for a required type `R` with a structural constraint, such

	Criterion	Definition
A)	Multi-type concepts	Multiple types can be simultaneously constrained.
	Multiple constraints	More than one constraint can be placed on a type parameter.
	Associated type access	Types can be mapped to other types within the context of a generic func.
	Constraints on associated types	Concepts may include constraints on associated types.
	Retroactive modeling	New modeling relationships can be added after a type has been defined.
	Type aliases	A mechanism for creating shorter names for types is provided.
	Separate compilation	Generic functions can be type checked and compiled independent of calls to them.
	Implicit argument deduction	The arguments for the type parameters of a generic function can be deduced and do not need to be explicitly provided by the programmer.
B)	Retroactive concept adaption	Concepts can be adapted after their initial definition. If the concept spans multiple types, a single adaption may affect several types.
	Retroactive constraint adaption	Constraints for generic parameters can be adapted and refined after their initial definition to better match existing code.

Table 2. *A) The different criteria for evaluation of support for generic constructs in programming languages, taken directly from [11, page 147]. B) Additional criteria for evaluation of support for adaptable generic programming. We discuss these criteria in further detail in Section 4.*

as "`inst T2 with R <= Runnable`", the compiler will check that the supplied type structurally conforms to the specification given by `R`. Conformance entails that all the required methods (and constructors if they are present, see below) must have an *exact* match (save for parameter names) in the supplied type. Covariant or contravariant signatures are not allowed; allowing this would not be type safe (e.g. when merging). Thus, for required types with only methods, the conformance relation for required types is equivalent to the <# matching relation of the $\mathcal{LOOM}$ language [5].

If the signature check succeeds, the compiler will *replace* all occurrences of `R` (based on the semantic analysis) in the instantiated template with the supplied type; in the example instantiation in the paragraph above, references to the type `R` will be replaced by references to the type `java.lang.Runnable`. The actual declaration of the required type `R` will be removed. Thus, for every class in the template, a version specific to this instantiation, with the given parameterization, is created. Note that, as opposed to in e.g. Scala, there is never a need for runtime reflection when dealing with structural constraints, since the actual type always will be known at *compile-time*.

Bounds for required types can be specified nominally as well as structurally. Taking the example from above, we can express that `R` must be a nominal subtype of e.g. the `Runnable` interface:

```
template T3 { required type R extends Runnable { } }
```

Thus, when supplying an actual type `A` for `R` in an instantiation of `T3`, it must explicitly implement or extend the `Runnable` interface (or `A` might be the `Runnable` interface itself).

Nominal and structural subtyping can also be mixed in a declaration of a required type. To demand an explicit, nominal, implementation of `Runnable`, and furthermore that a method `stop` must be present, we can easily express this as follows:

```
template T4 {
   required type R extends Runnable { void stop(); }
}
```

Classes and interfaces. Java generics do not allow the use of primitive types (though Scala does), and we do not in this work intend to lift that restriction. However, it is still important in some cases to be able to explicitly constrain a required type to be either an interface or a class.[3]

As can be seen from the syntactical overview in Figure 1, such constraints can be imposed by declaring a required type explicitly as either a `required interface` or a `required class`. Declaring requirements in this way puts further constraints on the required types; e.g. the ability to have constructors and fields are only available to required classes.

Thus, the term *required type* is overloaded in this paper, and is used both in the inclusive sense to refer to the syntactical and semantical constructs of required *types*, required *classes* and required *interfaces*, and, on the other hand, in the narrow sense to *only* refer to required *types*. When this distinction is important, we will be explicit about this; otherwise, the inclusive sense is implied.

Instantiation and concretization As mentioned above, upon instantiation of a template, the programmer may

[3] An example of a similar construct in a mainstream OO language is the `where R : class` constraints of C#.

required-spec	::=	`required` [<r-type> | <r-class> | <r-interface>]
r-type	::=	`type` <identifier> [`adds`] [<extends-clause>] { <r-type-body>* }
r-interface	::=	`interface` <identifier> [`adds`] [<extends-clause>] { <r-type-body>* }
r-class	::=	`class` <identifier> [`adds`] [<implements-clause>] [<extends-clause>] { <r-class-body>* }
r-type-body	::=	<method-signature>
r-class-body	::=	<constructor-signature> | <field-signature> | <method-signature>

Figure 1. *Syntax for required types. Non-terminals are written within <angled brackets>, and optional symbols are delimited by [square brackets]. A vertical line* (|) *signifies alternatives, and a star* (*) *signifies zero or more repetitions of a symbol. Terminal symbols are written with a* `monospace font`*. Productions left out (for the sake of brevity), such as e.g. the* extends-clause*, are to be understood as syntactically equal to their pure Java equivalents.*

choose to supply an actual concrete type for a required type `R` that satisfies the constraints of `R`. We will refer to this as a *concretization* of the required type.

A template can be instantiated in other templates and in packages. When a template `T` is instantiated in another template `U`, it is not mandatory to concretize the required types of `T`. Any required types in `T` that are not concretized upon instantiation in `U` will be propagated to `U`, and will thus become required types of `U`.

On the other hand, when a template `T` is instantiated in a package `P`, every (remaining) required type must be given a concretization. The concrete types may be classes or interfaces from instantiated templates (including the template containing the required type), or from other ordinary Java packages.

Sometimes, it can be nice to provide sensible default concretizations for required types, to alleviate the burden of always having to concretize every (remaining) required type when a template is instantiated in a package. For a mechanism like PT*r*, where a number of required types can be gathered in one template, a way to specify such defaults would indeed be helpful. A default concrete type or implementation could explicitly be given in the declaration of a required type. Another option for simple cases is to choose a default concretization from the bound of the required type. We are still studying how this can best be done, but for simple cases, the prototype compiler currently resorts to the latter approach.

Subtype hierarchies. Table 3 shows the relationships that are supported between required types, required interfaces, required classes, classes, and interfaces, for the `extends` and `implements` relations, respectively.

An extends or implements relationship between two required types does not in itself form a hierarchy. Rather, it puts forth a requirement for a hierarchy, i.e. a constraint that actual supplied types must (transitively / reflexively) fulfill.

Constructor definitions. Although it was not explicitly treated in [11], a seemingly common issue with type parameters is that you might want to create objects of the actual types. In Java and Scala, however, this is disallowed, so even if you have a method or a class parameterized by a type `T`, you cannot say "`new T()`".[4] C# is a bit more expressive, and allows the developer to constrain the type parameter by adding a "`where T: new()`" constraint, requiring the actual type to have a parameterless constructor. Other kinds of constructor requirements cannot be expressed.

When utilizing `required` classes, constructor requirements can quite naturally be handled simply by adding the necessary required constructor signatures to the class. These requirements can subsequently be structurally matched with the actual constructors of the class supplied upon instantiation. An example is shown below:

```
template T { required class E { E(int value); } ... }
```

`extends`	RT	RI	RC	I	C
RT			✓		
RI	✓	✓		✓	
RC			✓		
I	✓	✓		✓	
C			✓		✓

`implements`	RT	RI	RC	I	C
RT					
RI	✓		✓		✓
RC					
I	✓		✓		✓
C					

Table 3. *Support for the* `extends` *and* `implements` *relations between required types (RT), required interfaces (RI), required classes (RC), ordinary (template) interfaces (I) and ordinary (template) classes (C). The table is supposed to be read from the top row and down and to the left. I.e.,* `RC extends RT` *is a legal relationship, while the converse* `RT extends RC` *is not.*

[4] You can, however, in some cases create an object of a generic type `T` using reflection. Furthermore, in Scala, you can utilize implicit factories for similar results.

Inside classes in the template `T` above, or in classes from other templates or packages that instantiate `T`, statements such as "`new E(42)`" can safely be used.

Note that required constructors can *only* be defined for required *classes*, and not for required interfaces or plain required types.

For a required class `RC` that has a nominal subtyping requirement with bound `B`, where `B` is a class with accessible constructors, the required class must still structurally specify constructor requirements explicitly if "`new RC(...)`" is to be allowed. This is because a Java subclass in general needs not implement the same constructors as its superclass.

Refinement through additions. A required type in PT*r* can be given additions in the same way as classes and interfaces can in basic PT, through an `adds` clause. This can be used to refine constraints in subsequent instantiations. Consider a template `T` defined as follows:

```
template T { required type R { void run(); } }
```

In another template `U` that instantiates `T`, `R` can be refined by adding nominal or structural constraints. An example that does both is shown below:

```
template U {
  inst T;
  required type R adds implements Runnable {void stop();}
}
```

Here, `R` is *refined* in `U`, and further constrained to both nominally implement the `Runnable` interface and to implement a parameterless `stop()` method that returns `void`.

Merging. With basic PT, classes (or interfaces) from different template instantiations can be merged to form one new class. The details of the merge mechanism are beyond the scope of this article, the interested reader is referred to [3] for a more thorough exposition.

In PT*r*, required types can be merged in the same way that ordinary template classes and interfaces can. As for ordinary merges, different *kinds* of types cannot be "cross merged" with each other. I.e. a required interface can only be merged with other required interfaces, required classes only with other required classes, and required types only with other required types. The main difference from ordinary class or interface merging lies in the handling of conflicts. If, in the merge of two required types, there are equal signatures stemming from each of the types, this is not considered a conflict. Rather, the two signatures are merged into one in the resulting required type. If a given pair of equal signatures should indeed be kept separate, the developer may explicitly rename one or both of them in the instantiation. Merging required types where more than one has a nominal bound that is a class is considered a compile time error.

	C++	Java	Scala	PT*r*
Multi-type concepts	*	○	●	●
Multiple constraints	*	●	●	●
Associated type access	●	◐	●	●
Constr. on assoc. types	*	◐	●	●
Retroactive modeling	*	○	●	●
Type aliases	●	○	●	○
Separate compilation	○	●	●	◐
Implicit arg. deduction	●	●	●	●
Retroact. concept adapt.	○	○	◐	●
Retroact. constr. adapt.	○	○	◐	●

Table 4. *Support for adaptive generic programming in C++, Java, Scala, and PTr.*

Through merging of required types from different instantiations, the developer is able to express equality constraints across template instances, by explicitly declaring that two previously distinct required types are to be considered the same in the context of the current package or template. In contrast to an ordinary equality constraint, a merge also alleviates the need to provide the same parameter twice, making for more succinct code.

4. Fulfilling the Generic Programming Criteria

In this section, we discuss how and to what extent PT*r* fulfills the requirements listed in Table 2, as shown in Table 4. For brevity, we will not discuss scores that PT*r* "inherits" directly from Java.

Multi-type concepts. The essence of supporting multi-type concepts lies in the ability to simultaneously constrain more than one type. In PT*r*, constraints can be specified by required types within templates, to which several other (required) types of a multi-type concept can refer, and thus be simultaneously constrained. An example of this from our implementation of the generic graph library is shown below:

```
template GraphConcepts {
 required type Vertex { }
 required type Edge { Vertex source(); Vertex target();}
 required type EdgeIter extends Iterator<Edge> { }
 required type OutEdgeIter extends Iterator<Edge> { }
 required type VertexIter extends Iterator<Vertex> { }
 required interface IncidenceGraph {
   OutEdgeIter out_edges(Vertex v);
   int out_degree(Vertex v); }
 ...
}
```

As we can see from the code, the `Edge`, `VertexIter`, and `IncidenceGraph` types are all constrained by the (same) `Vertex` type, and the `EdgeIter` and `OutEdgeIter` types are constrained by the `Edge` type. Furthermore, we here see an example of traditional Java type pa-

rameterization (of the `java.util.Iterator<T>` interface) combined with PT*r*'s required types. Note that this template can be instantiated by relying on default concretization, as discussed briefly in Section 3, so that we might e.g. only explicitly concretize `Vertex` and `Edge`, and let the compiler concretize the remaining required types to their bounds. Note also that a concept can be composed from other concepts, each of which might span one or more types, by instantiating templates representing other concepts. Thus, with PT*r* one can express sub-concepts that are themselves comprised of other sub-concepts and/or types, and reuse and/or refine the constraints from these.

Multi-type concepts in Scala are typically implemented through use of the *Concept pattern* [22], parameterized with multiple type parameters. Applying this pattern thus implies creating separate concept classes (or singleton objects) that implement/model the concept interface/trait. Using Scala's `implicit` definitions this can in many cases make for a quite elegant solution. However, for the graph library functionality we found that having multiple multi-type concepts, implemented through the Concept pattern and constrained by the same associated/abstract types, quickly led to a rather complex solution.

Also in Java, multi-type concepts can be expressed through the Concept pattern, but this approach may quickly become cumbersome due to the fact that concept implementations must be referred to explicitly. The score for Java from [11] in Table 4 is instead based on a nominal subtyping approach.

Associated type access. Access to associated types is a property that allows code to refer to types that are associated with a generic concept. For instance, the general `Graph` concept has associated types `Edge` and `Vertex`. If the concept is expressed in a package template, and associated generic types as required types, one can simply refer directly (without any additional qualification) to the required types `Edge` and `Vertex`. These required types will be replaced by the actual types upon instantiation, at the latest in a package. Thus, associated type access comes "for free" with PT*r*.

In languages like Java, associated types are typically represented by generic parameters, and this quickly leads to verbose definitions. As an example, contrast the Java definition skeleton in Figure 2 of the breadth first search algorithm from [11] with the Scala and PT*r* versions directly below it.

Note how in the PT*r* version, the only parameterization that is necessary for the algorithm is to specify the `ColorMap` type, which is not in itself an associated type of the graph concept. The associated (required) type `Vertex` can be accessed directly (even though its actual type has not been supplied yet).

```
// Java version:
class breadth_first_search {
 public static <Vertex,
  Edge extends GraphEdge<Vertex>,
  VertexIterator extends Iterator<Vertex>,
  OutEdgeIterator extends Iterator<Edge>,
  ColorMap extends ReadWritePropertyMap<Vertex, Integer>>
  void go(VertexListAndIncidenceGraph<
         Vertex,Edge,VertexIterator, OutEdgeIterator> g,
      Vertex s, Visitor vis, ColorMap color) {
    ...
    graph_search.go(g,s,vis,color, ...);
} }

// Scala version
object breadth_first_search {
 def go[Graph <: VertexListAndIncidenceGraph,
    ColorMap <: ReadWritePropertyMap
          {type Key = Graph#Vertex; type Value = Int}]
       (g: Graph, s:Graph#Vertex, vis: Visitor,
          color: ColorMap ){
    ...
    graph_search.go(g, s, vis, color, ...);
} }

// PTr version:
inst GraphConceps;
class breadth_first_search {
 public static <ColorMap extends
     ReadWritePropertyMap<Vertex, Integer>>
  void go(VertexListAndIncidenceGraph g, Vertex s,
      Visitor vis, ColorMap color) {
    ...
    graph_search.go(g,s,vis,color, ...);
} }
```

Figure 2. *Associated type access in Java, Scala and PTr*

In Scala, an associated type is typically implemented as an abstract type within a class or a trait. Access to such a type is achieved through the type projection construct, e.g. `Graph#Vertex`. For the Scala code above, a parameter for the graph type is thus needed to access the `Vertex` type. Also note that neither the Java nor the Scala versions are parameterized on the `Visitor` concept, as this is not an associated type of the graph concept. For PT*r*, on the other hand, the `Visitor` concept is realized as a required type, and parameterization is thus available without any additional overhead.

A limitation that made implementing graph library functionality a little harder and less natural in Scala was the fact that you cannot use an abstract type as the type of a parameter to a method in another abstract type [21, sec. 3.2.7].

The example presented in Figure 2 is closely related to the issue of *constraint propagation*. We notice in that example that the PT*r* version does not need to mention the constraints for generic types it does not itself directly utilize, whereas the Java version must repeat the constrains for `VertexIterator`, `EdgeIterator`, etc. The example below shows how this can lead to complexity in even very simple cases:

```
// Java version:
interface VertexListAndIncidenceAndEdgeListGraph<
    Vertex,
    Edge extends GraphEdge<Vertex>,
    VertexIterator extends java.util.Iterator<Vertex>,
    OutEdgeIterator extends java.util.Iterator<Edge>,
    EdgeIterator extends java.util.Iterator<Edge>>
  extends
    VertexListAndIncidenceGraph<Vertex,Edge,
       VertexIterator,OutEdgeIterator>,
    EdgeListGraph<Vertex,Edge,EdgeIterator> {}

// PTr version:
required interface VertexListAndIncidenceAndEdgeListGraph
 extends VertexListAndIncidenceGraph,EdgeListGraph {}
```

The interface above, in either version, defines nothing more than a composition (through inheritance) of existing interfaces, and does as such not introduce any associated types or requirements on its own. The PT*r* version can be written in a much more succinct manner because there is no need to repeat the associated types as they are *propagated automatically upon instantiation*, and can thus be accessed without resorting to additional generic type parameters.

For Scala, the definition of `VertexListAndIncidenceAndEdgeListGraph` would be similar to the PT*r* version, but its constituents would in each of their definitions have to repeat the constraints for vertices and edges (and the concept could not itself be an associated type, due to the limitation mentioned above).

Constraints on associated types. There are several kinds of constraints that can be useful for associated types. A common form of constraints is that of an equality constraint, i.e. the requirement that an associated type of two other types must be the same. In a template with required types, this can easily be achieved in PT*r* by referring to the same requirement in both of the types. For associated types that were previously unrelated, one can express that they should in a given context be the same by merging the corresponding required types; the new required type will represent the union of the original ones.

Another typical form of constraints are in the form of sub/super relationships. With PT*r*, this can be expressed directly, with e.g. declarations of the form "`required interface I extends J {...}`". If `J` is itself a required interface, the actual type supplied for `I` is constrained to be a subtype of the actual type supplied for `J`.

Basic PT (and thus also PT*r*) amends the problem inherent to Java (and Scala) generics where it is not possible, due to type erasure, to constrain a generic type to two different parameterizations of the same generic interface (or trait). PT allows multiple instantiations (and thus also parameterizations) of a single template.

Retroactive modeling. In C++, retroactive modeling is implicitly supported since the compiler does not check the constraints put forth by templated concepts. Hence, any type can be said to model a concept without any prior reference to the concept itself, as long as the type provides the (implicitly) required operations.

In Java, there is no direct support for retroactively saying that a given class models (implements) a given concept (interface), short of changing its source code. However, a work-around might be the Concept pattern, though this is somewhat awkward to use since concepts must explicitly be passed around.

As Java, Scala supports retroactive modeling through the use of the Concept pattern, but `implicit` declarations make the use of concepts much more convenient and natural to the programmer in Scala.

Existing Scala libraries can be extended (or rather, appear to be extended) through the "library pimping" approach [20], in order to support retroactive modeling. However, this approach is typically based on implicit runtime creation of new objects, which might lead to subtle bugs e.g. when references to such objects are passed around. There is, in our opinion, a significant difference between retroactively adjusting the model (as PT*r* can do), and annotating the model with conversions to and from the modeled concepts.

With PT*r*, classes from templates can model new concepts through being merged with other classes that model the concept, or by having interface implementation declarations added by an `adds` part. The possibility for name changes makes it easier to let existing code retroactively model new interfaces. A small example sketch is shown below, where a concept `M` is realized by the interface `M`. The template `T` contains a class `C`, that implements the desired functionality in a method `mx`, however, it does not explicitly implement `M`:

```
interface M { void m(); }
template T { class C { void mx() { ... } } }
```

With PT*r*, we can retroactively define the implements-relation between `M` and an instance of `C`, as follows:

```
// rename method "mx" to "m":
inst T with C => C ( mx() -> m );
// add the interface implementation decl:
class C adds implements M { }
```

Type aliases. Type aliases are supported by C++ and Scala (and other languages) as a way to make long type names shorter, and are as such especially useful when dealing with heavily parameterized code. However, it has not been our goal to support this in our work with PT*r*, and the level of support is thus the same as for plain Java. Even so, PT*r* does alleviate this issue to a certain extent, since the parameterized types can be fixed at the time of instantiation, and need thus not be repeated for subsequent uses.

Separate compilation. This criterion includes both separate type checking and compilation into independent units. Java and Scala support both parts of the criterion, while the templates of C++ supports neither. The former part is fulfilled by PT*r*, since every template can be separately type checked independently of subsequent usage. The latter part is not supported by the current prototype compiler, which produces separate code for each instantiation (a heterogenous implementation). However, we have previously experimented with how a homogenous implementation can be made for an extended JVM, with special instructions e.g. for invoking methods in adapted template classes. Such an approach does, in contrast to the current heterogenous compile-time specialization scheme, come with some runtime performance overhead (which is also incidentally true for Scala's implicit definitions and Java's runtime casts due to erasure).

Retroactive concept adaption. As one of the additional two criteria we added in order to support the programming of adaptable generic libraries, retroactive concept adaption is the ability to unintrusively (i.e. without making changes to the original source code) make certain changes to a concept after its initial definition. These changes include renaming of methods and of the concept itself, and changes to the types returned by or accepted as parameters to the operations (methods) of the concept. Such changes can be important in order to provide a better match for existing code, or in order to better reflect the domain of the problems that the program is supposed to solve. Name changes may seem like a trivial modification to any program, but influential development methodologies like domain-driven design (see e.g. [9]), as well as research into naming conventions and intended semantics (see e.g. [13]), put emphasis on the importance of proper naming. Neither Java nor C++ supports the retroactive adaption of concepts, and developers are hence relegated to either make do with the concept as they were originally defined, make wrappers around them, or duplicate code. In Scala, concepts can, to a certain extent, be retroactively adapted through the use of implicit declarations and inheritance.

For PT*r*, retroactive adaption is one of the main motivating goals for the mechanism, and adaption of concepts (defined as interfaces, required interfaces or abstract base classes) as well as their potential implementations can be expressed as part of an instantiation of a template. The fact that each instantiation results in a new set of classes is the main reason for PT*r*'s flexibility with regards to name changes.

Beyond name changes, a concept in PT*r* might be refined by adding new operation signatures through the use of `adds` clauses (for both concept definitions and implementations). The PT*r* approach supports overloads and (single) dynamic dispatch of added operations, in contrast to mechanisms such as extension methods in C# or the Concept pattern through implicits in Scala, which rely on static dispatch.

Retroactive constraint adaption. Constraints form a significant part of the interface to a generic library, and hence if retroactive modeling and adaption are deemed important, the possibility for retroactive adaption of constraints should be of equal importance.

In Java, constraints cannot be adapted short of changing the source code or creating wrappers that refine the original constraint. Scala supports refinement of constraints expressed as abstract types through subtyping, but the original constraint cannot be adapted.

With PT*r*, constraints in form of required types can be adapted in several ways. To begin with, their names can be changed, along with the names of method signatures within them. Changing the name of a required type might be useful when a type is not supplied at instantiation, and a default interface definition is subsequently created by the compiler. Changing the names of method signatures is useful in order to adapt the generic library to existing code, when one is unable or unwilling to change the latter. Below is an example of a small library for representing cities and roads, encapsulated in a package named `Geography`.

```
package Geography {
  class City { String name; int population; ... }
  class Road {
    private City from, to;
    City getFrom() { return from; }
    void setFrom(City c) { from = c; }
    City getTo() { return to; }
    void setTo(City c) { to = c; }
} }
```

It would be nice to be able to apply the algorithms from the generic graph library, like e.g. Dijkstra's shortest paths, to cities and roads from the `Geography` package. In our implementation, the `Algorithms` template contains the desired functionality, and it will in turn instantiate the `GraphConcepts` template, which contains the requirement for a `Vertex` class and an `Edge` class, and corresponding constraints. The requirements will be propagated to the instantiating package. Thus, we can use the instantiation below to adapt the generic constraints to our `Geography` package:

```
inst Algorithms with Vertex <= City,
   Edge (source() -> getFrom, target() -> getTo) <= Road;
```

With this approach, we can apply all the algorithms from the generic graph library to the classes from the `Geography` package.

A constraint may also be adapted in PT*r* by providing an addition that refines the constraint, typically in a narrowing fashion. For instance, to further constrain

the `Vertex` type to include a `getName` method, we can utilize the following code in a template:

```
inst GraphConcepts;
required type Vertex adds { String getName(); }
```

It is important to note that such modifications are local to the current instantiation, and do not propagate globally to other potential instantiations of the `GraphConcepts` template in other parts of the program. Thus, retroactive constraint adaption can in PT*r* be done in a controlled and unintrusive manner.

Aside: code complexity comparison. The Java implementation from [11] has 760 lines of code[5], while the corresponding PT*r* implementation has 691 lines. However, this includes a lot of imperative code that is identical in the two versions. To approach an understanding of the relative complexity in terms of parameterizations and constraints, we have tried to count these in a reasonable way. In both versions we have counted all elements that occur within angle brackets <. . . >, and all elements that occur as subtype bounds for generic constraints. In addition, we have counted all required types and explicit concretizations of such for PT*r*. The count is 542 occurrences for Java and 325 for PT*r*, i.e. a reduction of about 40%. We think that this can have a significant impact on the comprehensibility and maintainability of the code.

5. Related Work

Virtual classes originated with the BETA language [17, 18], and has subsequently inspired a host of other languages and mechanisms, such as e.g. gbeta [7], Caesar [1], J& [19], and Newspeak [4]. These mechanisms support a certain degree of parameterization based on overrides (or *refinements*) of the virtual types, typically contained within ordinary classes. An important advantage of the virtual type approach over type parameterization as found e.g. in Java is the automatic propagation of constraints.

In [26], Thorup argues for the inclusion of virtual types in Java as an alternative to classes with type parameters. In this proposal, Java classes can contain bounded `typedefs`, that can be used to define generic classes that abstract over an open set of types. A problem with this approach is that static type safety is reduced, and every class could now potentially be subject to runtime exceptions due to covariant subtyping. With PT*r*, this kind of typing issues are not problematic since actual types *substitute* all occurrences of references to required types at *compile-time*.

A subsequent paper [27] presents an approach that combines virtual types (in a type safe variant [6]) with structural subtyping. This gives three "dimensions" of subtyping: the ordinary subclass variant, covariance in the generic parameter, and binding of the generic parameter to its bound. This allows virtual types to be used in many situations where parameterized types traditionally have been considered a better option. We have not discussed the issue of subtyping of parameterized classes to any extent in this paper, but it seems clear that if subtype relations between different parameterizations of the same class are required, PT*r* is not the ideal tool, since every instantiation results in a new, independent, set of classes (though template classes can implement common external interfaces).

Both [26] and [27] provide adequate support for associated types of concepts, however, as opposed to in PT*r*, multi-type concepts are not as easily expressed.

In [14], the authors introduce explicit support for associated types and constraint propagation in C# with a mechanism resembling virtual classes. However, an important distinction from prototypical virtual classes is that nested types are not specific for each object of an outer class. Like PT*r*, they support assignment of preexisting types to the associated (virtual) type definitions. Constraint propagation is automatic, as for PT*r*, but limited to the confines of singular class hierarchies.

Neither of the virtual type-based mechanisms support retroactive adaption of concepts or constraints in a manner resembling PT*r*.

Scala [21] has been discussed in some detail in this paper, however, there are some additional points that should be addressed. We have mentioned that the static nature of our mechanism facilitates a simpler type system, and arguably also a simpler conceptual model. However, this has some obvious drawbacks, most notably that full family polymorphism [8] and dependent types are not supported (since we do not allow creating instances of templates at runtime). Furthermore, Scala supports several advanced features that PT*r* does not, such as e.g. higher-ordered types, pattern matching and implicits.

JavaGI [28] is an extension to Java inspired by Haskell's type classes. It supports retroactive modeling through explicit implementation declarations. *Multi-headed interfaces* provide support for multi-type concepts in a natural way, exemplified by the Observer pattern [10]; a corresponding implementation in PT could be a template with types for each role, an example can be found in [2]. JavaGI does not fully support retroactive adaption.

The $\mathcal{G}$ language [24] compiles to C++ and contains explicit support for generic concepts and models, and supports all the criteria listed in Table 2A. Like PT*r* it supports modular type checking, and also separate compilation, but not the criteria from Table 2B.

[5] Counted with the CLOC tool: `http://cloc.sourceforge.net/`

Partly building on the work on $\mathcal{G}$, ConceptC++ [12] supports explicit concept definitions and constrained function and class templates. Retroactive modeling is achieved through concept maps, and in general the criteria from Table 2A are well supported, though type checking is not fully modular due to their support for concept-based overloading, which PT*r* does not fully support. A goal for ConceptC++ was to form the basis for the inclusion of concepts in the new C++0x standard. However, the current version of the standard excludes concept support.[6] To our knowledge, neither ConceptC++ nor C++0x fully supports the criteria in Table 2B.

6. Concluding Remarks

The package template (PT) mechanism extended with required type specifications yields a, to the best of the authors' knowledge, rather novel blend of support for parameterization and retroactive modeling and adaption through compile-time specialization with separate type checking. We refer to this variant of PT as PT*r*.

We have shown that PT*r* applied to a mainstream OO language like Java supports almost all of the criteria put forth by [11], as well as additional criteria identified in this paper for adaptability of generic code. Furthermore, early investigations suggest that PT*r* can provide a significant reduction of duplicated code in generic Java libraries, and in some cases provide a simpler solution compared to other mechanisms aiming at similar problems, such as virtual class- or abstract type-based mechanisms.

Acknowledgments

This work has been done within the context of the SWAT project (NFR grant 167172/V30). We thank the reviewers for helpful comments on previous versions of this paper, and Steinar Kaldager and Daniel Rødskog for dedicated work on the PT(*r*) compiler.

References

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In *Trans. AOSD I*, volume 3880 of *LNCS*, pages 135–173. Springer Berlin / Heidelberg, 2006.

[2] E. W. Axelsen, F. Sørensen, and S. Krogdahl. A reusable observer pattern implementation using package templates. In *ACP4IS '09*, pages 37–42, New York, NY, USA, 2009. ACM.

[3] E. W. Axelsen, F. Sørensen, S. Krogdahl, and B. Møller-Pedersen. Challenges in the design of the package template mechanism. *Transactions on Aspect-Oriented Programming*, 2012. To appear, available now from `http://swat.project.ifi.uio.no/`.

[4] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in newspeak. In T. D'Hondt, editor, *ECOOP 2010*, LNCS. Springer, 2010.

[5] K. B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good match for object-oriented programming languages. In *ECOOP '97*, 1997.

[6] See e.g. `http://drdobbs.com/architecture-and-design/218600111?pgno=3`

[6] M. T. Computer and M. Torgersen. Virtual types are statically safe. In *Proc. FOAL '98*, 1998.

[7] E. Ernst. gbeta - a language with virtual attributes, block structure, and propagating, dynamic inheritance, 1999.

[8] E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP*, volume 2072 of *LNCS*. Springer, 2001.

[9] E. Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321125215.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns -Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[11] R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17:145–205, March 2007.

[12] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *Proc. OOPSLA '06*, pages 291–310, New York, NY, USA, 2006. ACM.

[13] E. Høst and B. Østvold. Debugging method names. In S. Drossopoulou, editor, *ECOOP 2009*, volume 5653, pages 294–317. Springer Berlin / Heidelberg, 2009.

[14] J. Järvi, J. Willcock, and A. Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. OOPSLA '05, pages 1–19, New York, NY, USA, 2005. ACM.

[15] M. Jazayeri, R. Loos, and D. Musser. Generic Programming - Report from Dagstuhl Seminar. Technical report, 1998.

[16] S. Krogdahl, B. Møller-Pedersen, and F. Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. *Journal of Object Technology*, 8(7):59–85, 2009.

[17] O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89*, New York, NY, USA, 1989. ACM.

[18] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley, New York, NY, USA, 1993.

[19] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06*, pages 21–36, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4.

[20] M. Odersky. Pimp my library, 2006. URL `http://www.artima.com/weblogs/viewpost.jsp?thread=179766`.

[21] M. Odersky. The scala language spec. version 2.9 – draft, 2011.

[22] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *Proc. OOPSLA 2010*, pages 341–360, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6.

[23] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 327–339. Springer Berlin / Heidelberg, 2003.

[24] J. G. Siek and A. Lumsdaine. A language for generic programming in the large. *Science of Computer Programming*, 76(5):423 – 465, 2008. ISSN 0167-6423.

[25] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual (C++ In-Depth Series)*. Addison-Wesley Professional, Dec. 2001. ISBN 0201729148.

[26] K. K. Thorup. Genericity in java with virtual types. In *In Proceedings ECOOP '97*, pages 444–471. Springer-Verlag, 1997.

[27] K. K. Thorup and M. Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. ECOOP '99, pages 186–204, London, UK, 1999. Springer-Verlag.

[28] S. Wehr, R. Lämmel, and P. Thiemann. JavaGI: Generalized Interfaces for Java. In *ECOOP 2007, Proceedings*, LNCS. Springer-Verlag, July 2007.