Controlling Dynamic Module Composition through an Extensible Meta-Level API

Eyvind W. Axelsen Stein Krogdahl Birger Møller-Pedersen
University of Oslo, Department of Informatics
Postboks 1080 Blindern, 0316 Oslo, Norway
{eyvinda, steinkr, birger}@ifi.uio.no

Abstract

In addition to traditional object-oriented (OO) concepts such as inheritance and polymorphism, several modularization and composition mechanisms like e.g. traits, mixins and virtual classes have emerged. The Package Template mechanism is another attempt at providing a flexible mechanism for modularization, composition and adaption.

Dynamic languages have traditionally employed strong support for meta-programming, with hooks to control OO concepts such as method invocation and object construction, by utilizing meta-classes and meta-object protocols.

In this work, we attempt to bring a corresponding degree of meta-level control to composition primitives, with a concrete starting point in the package template mechanism as developed for the dynamic language Groovy. We wish to support a wide range of possible composition semantics, and to make such choices available to the developer through a meta-level API. This API should be extensible, and the semantic variations should be applied within well-defined scopes.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects; D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages

General Terms Languages, Design

Keywords OOP, Modularization, Dynamic Languages, Templates, Meta-Level

1. Introduction

Having good tools for modularization, reuse and adaption of code is a recognized need in the domain of software development. However, it seems that the traditional object-oriented mechanisms often tend to fall short when it comes to dealing with high degrees of complexity paired with a desire for the right degree of modularity (for the problem at hand). Several approaches have been introduced to better cope with such issues, such as e.g. traits [28], mixins [5], and mechanisms based on virtual classes, e.g. [1, 4, 9, 22, 23, 25], to name a few.

Package templates (PT) have been introduced as another approach to this for statically typed languages like Java [12] in [19], and subsequently for dynamic languages like Groovy [13] in [2]. Through the use of package templates, a library developer may write entire packages in the form of (package) templates that, upon subsequent usages, may be adapted and customized to fit a variety of different application and/or business domains.

However, a common property of all of these mechanisms is that they support only a rather strictly fixed set of composition semantics, and that there are limited options for extending or modifying this set (short of changing the source code itself, which often is an undesirable option). For a given use case, the chosen semantics of a mechanism might be undesirable, irregardless of the careful and thorough design that went into it. For instance, for traits the "composition order is irrelevant" [28, page 3], while for instance in the JavaFX's [10] implementation of mixins [7], a "mixin declared earlier in the extends list takes precedence over mixins declared later in the extends list" with respect to conflicting methods and fields; yet it is not clear that one variant is inherently better than the other for all usages. Obviously, the list of possible semantic variations in a language is virtually without bounds. Thus, *a mechanism that attempts to provide a greater degree of flexibility should in itself be extensible*.

For languages that support a meta-object protocol (MOP) [17], the application developer may control the semantics of method lookup and dispatch (beyond traditional virtual methods), field access etc. We propose that also for module import, adaption and composition, semantic control should be available to the programmer, and hence that the language's semantics should be open for modification through hooks or other fitting abstractions throughout the composition process. Such modifications to the semantics should have a well-defined, and optionally local, scope. Furthermore, through utilizing the MOP, many languages, including Groovy, allow objects to replace their own meta-object, and as such take control over their own semantics. Correspondingly, we propose that modules should be able to include code that utilizes meta-level functionality in order to programmatically control their own usage scenarios.

In this paper, we will focus on modules in terms of packages, and we propose a meta-level mechanism that allows the developer to interact with the runtime process of importing, adapting and composing such packages. Furthermore, the developer will be given the means to extend the mechanism in useful ways, such as e.g. adding new keywords with AOP-like functionality. Our approach is based on an extension of the dynamic package template mechanism for the Groovy language (which we shall refer to as GPT) [2]. While the examples will be specific for Groovy and GPT, we believe that the mechanism and the problems discussed in themselves are, at least to a certain extent, general, and that it should be viable in and for a wide range of languages and composition mechanisms (though admittedly the language in question's support for reflection and the nature of the type system, be it static or dynamic, will be of great importance with regards to the ease of such an implementation).

An added motivation for this work is also to more fully grasp and reap the potential of the combination of dynamic languages like Groovy, and strong modularization mechanisms like PT; it seems that with the GPT basics in place, there is a wide range of possible and hopefully fruitful extensions that can be applied to further improve the expressiveness and flexibility of the mechanism, in keeping with tradition for such languages.

The contributions of this paper are: we present an extensible meta-level approach for controlling the semantics of software composition, and demonstrate the supposed validity of this approach on a few non-trivial examples. We also provide a proof-of-concept implementation in the form of a pluggable jar file for use in any Groovy program.[1] The main difference between our approach and other works on package/module composition is that we provide an open and extensible API that, in addition to requiring no extra artifacts in form of e.g. new compilers or composition rule languages, also opens up for a host of new ways to interact with the composition process.

The remainder of this article is structured as follows: In Section 2, we briefly describe the distinguishing features of the Groovy language, and the main concepts of PT for Groovy. (For a thorough treatment of the same concepts for PT for statically typed languages, the interested reader is referred to [19].) Section 3 describes the main concepts of our meta-level API. A set of motivating and detailing examples is presented inline with the different topics of this section. A brief discussion on the generality and applicability of this work can be found in Section 4. The implementation of the mechanism is briefly discussed in Section 5, and related work is discussed in Section 6. Section 7 concludes this article and touches on possible ideas for future work.

2. Background

2.1 Groovy

Groovy is a dynamic language that runs on the Java Virtual Machine. It can hence integrate nicely with Java, and to a certain extent it provides a familiar Java-esque syntax. The language rests on a purely object oriented foundation, and provides support for several attractive language concepts.

Both static and dynamic typing of variables and method signatures is supported in Groovy. However, there is presently no performance to be gained in Groovy by using static typing as opposed to dynamic (and in many cases it may actually be slower). Furthermore, Groovy cannot detect e.g. invalid calls to statically typed methods at compile time (since a fitting method may be added to the object in question at runtime) [14]. We will therefore not consider the static typing features of Groovy in this paper, and instead focus solely on dynamically typed references, methods and properties defined by using the keyword `def`.

A runtime meta-object protocol (MOP) is supported by the Groovy language. In accordance with the MOP, every object and class has an associated object of a meta-class. This meta-class object may be replaced by a version defined by the programmer, and this allows the semantics of method invocation and property getters and setters to be changed on a per-object or per-class basis. Also, new members may be added to classes and objects at runtime using the meta-class.

[1] The prototype implementation can be downloaded from `http://home.ifi.uio.no/eyvinda/dls10`.

Further tailoring of the language's semantics may be done using compile-time transformations of the abstract syntax tree (AST). AST transformations may be defined by the developer to transform or replace parts of the syntax tree during different phases of the compilation process, such as parsing, semantic analysis, code generation etc. Such transformations may be global, or local pertaining to certain targets that typically may be Java-like annotations on source code elements. Dynamic typing aside, one thing in Groovy that is resolved entirely at compile-time is the given class from which to create a new object (e.g. as in `new A()`). That is, if a referenced class `A` cannot be statically determined to exist within imported packages and with correct accessibility modifiers, a compile-time error will occur. Since we will be loading packages dynamically, we will use compile-time AST transformations to get around this issue.

Closures are first class citizens of Groovy, and units of code may as such be passed to and from methods. Closures are hence also objects, and have their own meta-class object that may be replaced by the programmer. It is worth noting, however, that Groovy closures are not actually closures in the usual sense of the word, since they do not necessarily *close* around the variables to which code within the closure refers (i.e., closures in Groovy can contain unbound free variables). We shall nevertheless in this paper use the term *closure* to refer to the syntactic constructs that define subclasses of `groovy.lang.Closure`, and instances thereof.

Closures might have a *delegate*, to which method calls and field accesses can be dispatched (by setting the closure's resolution strategy to for instance `Closure.DELEGATE_FIRST`). The delegate is, as the meta-class object, replaceable by the developer.

A closure is written within curly braces, and may take parameters (or be parameterless). Consider a small example adapted from [13]:

```
def c = 1
def printSum = { a, b -> print a + b + c }
printSum(5, 7) // prints 13
```

Now, by setting `printSum.delegate` to an object that contains a field named `c` with a value that is different from 1, and setting the resolution strategy as mentioned above, we can change the printed output, and this, together with MOP features, provides for flexibility that proves to be very useful when implementing an open and extensible API such as the one we are about to describe.

Parentheses and semicolons in method calls can be omitted in Groovy. Thus, for a method `m` that takes a closure as its sole argument, the call can be written as `m { ... }`, where the ellipsis represents the actual code of the closure.

2.2 Groovy Package Templates at a Glance

As mentioned in the introduction, the package template mechanism was introduced in [19]. In that paper, Java was the target language for the mechanism, and a detailed exposition of the features of the original version of the mechanism can be found in that paper. We later adapted the main concepts of the mechanism to dynamic languages, with particular focus on Groovy in [2], and it is the latter version that will be the basis for this paper. Below we explain the basic concepts of the mechanism (most of which indeed are common for both static and dynamic versions), before we move to the meta-level functionality in the following sections.

The package template mechanism targets the development of reusable and adaptable collections of interdependent classes. A package template looks much like a regular Groovy file, with the most notable exception being a `template_def` construct enclosing classes and interfaces. An example is shown below:

```
template_def T {
   class A { ... }
   class B extends A { ... }
}
```

The contents of a template are ordinary Groovy classes or interfaces. These types may refer to each other and take part in type hierarchies within the template. A template class is not allowed to have inheritance relationships to classes defined in packages outside the template (with the exception of classes in package templates that are instantiated within the current template, more on instantiations below). Templates need to be semantically (and, of course, syntactically) valid in their own right, and may as such be checked for validity according to the rules of the target language (in this case Groovy) independently of their potential usage.

None of the classes or interfaces defined in a template are available to the rest of the program before the template itself is instantiated (that is, the class loader does not see any of the members of a template before an instantiation of the template has been performed).

A package template must hence be explicitly instantiated before any of its contents can be used. A template can be instantiated multiple times, and each instantiation of a given template is completely independent of any other instantiations. The result of an instantiation is a new package (we shall, in this paper, use the term *package* in a broad sense, meaning a collection of classes and interfaces that is internally consistent, but not necessarily implemented as a regular Java or Groovy package as such). A package template is instantiated at runtime by utilizing the `instantiate` method of the framework's PT class, as shown in the following construct:

```
def INST_T = PT.instantiate {
   template T {
      map B, BB
} }
```

This statement[2] will make *a copy of* all the classes and interfaces defined in the template T available to the caller of the static instantiate method through the variable INST_T[3], preserving potential type hierarchies as defined in the template. Classes within the new package can be accessed by qualifying their name with the variable name, e.g. as in `new INST_T.A()`.

The sub-statement "map B, BB" entails that the class B (and all explicit references to that type) should be renamed (retyped) to BB.

However, instantiations may also be more elaborate: the user can specify that classes or methods should be renamed, that classes should have additions, and/or that (previously unrelated) classes should be merged.

To facilitate this, implementation-wise, the PT.instantiate method takes a Groovy closure as its sole argument, and this closure may in turn contain several calls to the template method. The template method takes a template name (such as T in the example above), and optionally a specification (in the form of a closure) of *adaptations*, such as mappings to new names etc. Thus, the instantiation specifications provided to the instantiate method can be viewed as expressions in a tiny internal DSL for instantiating package templates.

If multiple templates are listed in the argument provided to instantiate, this will lead to *one single* instantiation of these templates, forming *one single* new package.

Consider now a second, equally simple, template U:

```
template_def U {
   class C { ... }
}
```

Taking the example templates T and U from above, we can employ the following call to merge the classes A and C to form one new class AC:

```
def INST_TU = PT.instantiate {
    template T { map A, AC }
    template U { map C,  AC }
  }
```

Here, the classes A and C are both renamed (or *mapped*) to AC (using the map method), and they will thereby be merged according to the semantics of the mechanism (which, roughly, comes down to taking the discriminated union of all the attributes from each class involved in the merge). A new, identical (from a textual point of view), copy of the class B will be available through INST_TU, while the template classes A and C will not be available as individual classes, but only as their merge. The class B will now be a subclass of the new class AC, and all existing references to A within T and C within U will be retyped to AC. Already at this point, we clearly see that there are several possible options for semantic variations in the language, with respect to merge order, conflict resolution semantics etc.

PT also allows for additions to be made to instantiated classes, so that methods may be overridden and new methods and fields may be provided. In GPT this is done by writing an ordinary class with the same name as the one to which additions should be made, and decorating it with a @PTAddition annotation. The concept of additions is important in the (G)PT mechanism, but we will not make any further use of it in this paper, and hence we will not discuss it in any further detail. (The interested reader is referred to [2, 19] for a more detailed exposition.)

3. The Meta-Level Package Instantiation API

In this section, the main body of our work will be presented. We will start with a general overview of the main concepts of the mechanism and its application and scope in Sections 3.1 through 3.3. Then, in Section 3.4, we will look at how this can be applied to provide different options for resolution of potential conflicts in the composition process, by utilizing the possibilities of the meta-level API. As promised by the title of this paper, the API itself is also *extensible*, and in Section 3.5 we will explain how this can be utilized to provide convenient access to new functionality, exemplified through an (admittedly simple) AOP-like construct. Another main idea of this work is that units of code, such as package templates, can contain introspective meta-level code in order to control and customize their usage; this will be described in Section 3.6. Interactions between different meta-level pieces of code are briefly treated in Section 3.7.

[2] As mentioned in Section 2.1, the syntax of Groovy is quite flexible, allowing the developer to omit parenthesis and semicolons where desirable. Furthermore, the GPT framework will manipulate the resulting expression a little at runtime, to allow the developer to omit commas and quotation marks. The instantiate statement could hence also be written in a more elaborate (and perhaps more immediately familiar) way: `def INST_T = PT.instantiate({ template("T", { map("B", "BB")}) });`.

[3] The variable name holds no significance, but as a convention we will name variables holding instantiated templates INST_ followed by the template name(s) or a derivation thereof.

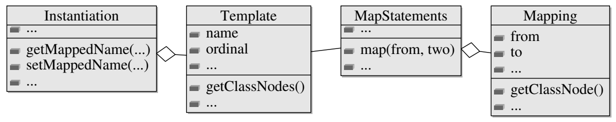

Figure 1. A slightly simplified view of the classes representing an instantiation of package templates.

3.1 Overview

When a module/package is utilized within a program, there needs to be a specification of how the module should be made available. Such specifications may have varying degrees of expressiveness and variation points. For e.g. a regular package in the Java language, a rather simple specification in form of the `import` statement is used. Common use cases are to specify that all the classes in a given package should be imported, e.g. `import javax.swing.*`, or that a specific class should be imported, e.g. `import javax.swing.-JOptionPane`. The modifier `static` may also be applied to allow static class fields and methods to be referenced without qualification.

For both the static and dynamic versions of PT, the developer can write specifications that describe the instantiation of package templates to form packages that are subsequently imported. These specifications can specify a significantly wider range of options compared to their Java counterpart (e.g. additions, merging, renaming etc), but the set of options is still fixed.

In the following, we will present a meta-level API for interacting with the instantiation and composition of templates and their contained classes. This API can be seen as a meta-level instance of the *strategy design pattern* [11], in which the provided semantics of the GPT mechanism as described in [2] can be seen as a base case, and different strategies may be supplied for different semantics. An alternate view is to look at this from an *open implementation* [16] point of view, in which the methods of the GPT framework from [2] would be the primary interface, and the API presented here would be the meta-interface. The functionality exposed through the primary interface would then be open for semantic customization through the meta-interface. Thus, the primary interface for a given package abstraction will to a certain extent dictate what is reasonable to include in the meta-interface.

In our concrete instance with GPT, the API will be closely tied to runtime calls to the `instantiate` method of the `PT` class. When the `PT.instantiate` method is called, as exemplified in the Section 2.2, a tree-like object structure representing the desired instantiation, as specified by the closure supplied as the actual parameter of the call, is created. A simplified view of the classes involved in this tree is shown in Figure 1. An object of the `Instantiation` class will be the root of the tree; it contains a list of templates to be instantiated, represented by objects of the `Template` class. `Template` objects contain a collection of `Mappings`, that are held by a reference to an object of the `MapStatements` class. This representation of an instantiation is an important part of the API, as it will be passed on to most of the methods invoked at the meta-level (see next section for details). This enables the developer to examine and manipulate the instantiation specification in order to control the semantics that are applied when creating the resulting package.

3.2 Meta-Level Interception Points and Strategies

In the same way that meta-object protocols typically provide *hooks* into e.g. method dispatch and object creation, our mechanism needs to provide sensible hooks into the package template instantiation and composition process. We will refer to such hooks as *interception points*. An obvious such point at which interception seems fruitful is when the object tree representing the instantiation specification (as objects of the classes in the class diagram in Figure 1) has been created. At this point, the specification (including which templates to instantiate, which classes to merge, etc.) might be changed by meta-level code.

The GPT framework provides access to the meta-level API by allowing the users to define classes that implement one of a set of interfaces provided by our framework. One such interface is the `PTInstSpec-Strategy`, shown below:

```
interface PTInstSpecStrategy {
   // The following method will be called once,
   // when a representation of an instantiation
   // specification has been created:
   def processInstantiationSpec(instantiation);
}
```

We shall refer to implementations of interfaces such as the one above as *strategies*. The flowchart in Figure 2 shows a conceptual view of the template instantiation process performed by our framework. First, the framework checks for so-called *setup strategies* (`A`). After the potential application of any such strategies, a representation of the instantiation specification is built. Then any strategies implementing `PTInstSpecStrategy` are applied, at the interception point marked `B` in the figure. (We shall return to point `A` in greater detail later on, in Section 3.5.)

Subsequently, an abstract syntax tree is built for each template specified by the instantiation. At this point, the instantiation specification has been processed (and is thus considered complete). The framework then checks for supplied implementations of the `PTPreInstStrategy` interface, shown in the code snip-

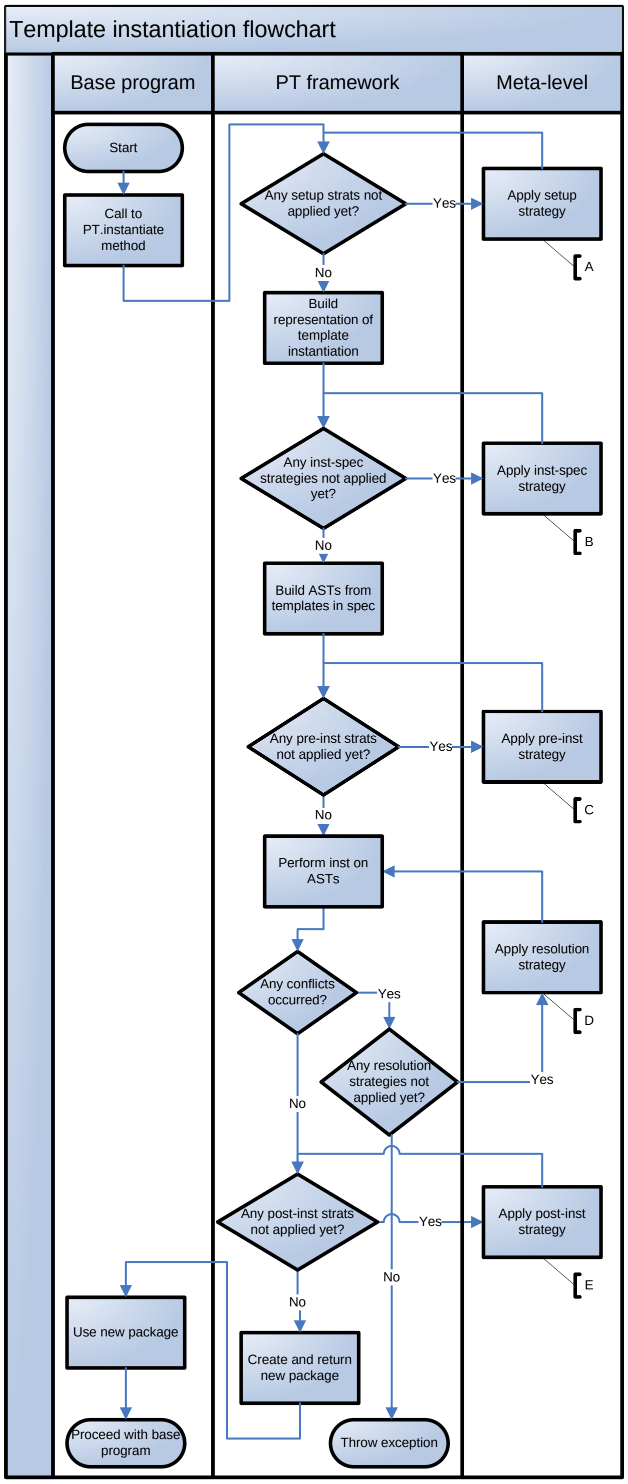
Template instantiation flowchart
Base program
PT framework
Meta-level
Start
Call to PT.instantiate method
Any setup strats not applied yet?
Yes
Apply setup strategy
A
No
Build representation of template instantiation
Any inst-spec strategies not applied yet?
Yes
Apply inst-spec strategy
B
No
Build ASTs from templates in spec
Any pre-inst strats not applied yet?
Yes
Apply pre-inst strategy
C
No
Perform inst on ASTs
Apply resolution strategy
Any conflicts occurred?
Yes
D
Any resolution strategies not applied yet?
Yes
No
Any post-inst strats not applied yet?
Yes
Apply post-inst strategy
E
No
No
Use new package
Create and return new package
Proceed with base program
Throw exception

Figure 2. A flowchart showing a conceptual view of the template instantiation process. The available meta-level interception points are labeled from `A` to `E` in the right-hand column in the chart. For each interception point, strategies are (by default) applied in the order they are listed, see Section 3.3 for details.

pet below, to allow changes to the template ASTs before the specified (and possibly modified) instantiation is carried out.

```
interface PTPreInstStrategy {

   // called for each template class:
   def processClass(classNode, template,
      instantiation);

   // called for each template interface:
   def processInterface(interfaceNode, template,
      instantiation);

   // called for each specified class merge:
   def processMerge(classes, instantiation);
}
```

Implementing strategies according to this interface allows for modification of classes (e.g. adding, renaming or removing methods) before the instantiation is carried out on the AST (i.e. renames, merges etc has not yet taken effect). A corresponding interface `PTPostInstStrategy` is also available, allowing for modifications to the AST after the instantiation specification has been performed, as shown below:

```
interface PTPostInstStrategy {

   // called for each instantiated class
   // (resulting from a merge or not):
   def processClass(classNode, instantiation);

   // called for each instantiated interface
   // (resulting from a merge or not):
   def processInterface(interfaceNode, instantiation);
}
```

Note that at the time that implementations of `PTPostInstStrategy` are invoked, any specified merges have already been performed on the AST, and the interface thus has no `processMerge` method, as opposed to the pre-instantiation version.

In between the pre and post instantiation steps, the actual instantiation is carried out on the AST. During this process, a set of conflicts that prevent the instantiation from being completed successfully might arise. Situations such as this can be handled at the meta-level at interception point `D` in Figure 2, with implementations of the `PTConflictResolutionStrategy` interface, as shown below:

```
interface PTConflictResolutionStrategy {

   // invoked upon each method conflict
   // resulting from a merge:
   def onMethodConflict(classNode1, methodNode1,
      classNode2, methodNode2, instantiation);
```

```
  // invoked upon each constructor conflict
  // resulting from a merge:
  def onConstructorConflict(classNode1, constrNode1,
     classNode2, constrNode2, instantiation);

  // invoked upon each field conflict
  // resulting from a merge:
  def onFieldConflict(classNode, fieldNode1,
     classNode2, fieldNode2, instantiation);

  // invoked for merges that would lead to
  // multiple inheritance
  def onSuperclassConflict(classNode1, classNode2);

  // corresponding signatures for conflicts
  // within merged interfaces here...
}
```

By creating classes implementing PTConflictResolutionStrategy, the developer is free to design alternate conflict resolution semantics; we shall take a closer look at this in Section 3.4.

As we have seen, the methods of the interfaces presented in this section take formal parameters named classNode, methodNode, fieldNode etc. The GPT framework will supply the actual parameter values at runtime, by creating objects of classes named PTClassNode, PTMethodNode, PTFieldNode etc. (PTClassNode is also the default class for the objects returned from the getClassNode(s) methods of the Template and Mapping classes from Figure 1.) These classes contain convenience methods with which the developer can interact in order to modify the AST of template classes and interfaces, as will be illustrated in the examples throughout the rest of Section 3.

The PTClassNode class contains the following methods, for which a sufficient understanding of their semantics should be inferable by their name: getName, setName, getMethods (which will return a collection of objects of the PTMethodNode class with standard collection operations for adding and removing objects), getFields, getConstructors, getSuperClass, getInterfaces and a protected getAST. The PTMethodNode class contains methods getName, setName, getSignature, setCode and a protected getAST. The setCode method probably deserves a comment; it takes a Groovy closure as its single formal parameter, and uses that for specifying both the signature of the method and the actual code of the operation. The method will be injected in the new class through the use of Groovy's meta-object protocol. An example usage of setCode will be shown in Section 3.4. For brevity we will skip listing the operations on the PTFieldNode and PTConstructorNode classes. However, it is worth noting that through the extension mechanisms described in Section 3.5, and by utilizing the getAST methods, one can extend the set of available operations for each of these classes.

3.3 Strategy Application and Scope

There are at least two valid scopes in which a strategy could apply, and that is global, i.e. for every instantiation throughout a program, and local, pertaining only to one specific instantiation. For the global scope, the PT class provides an addStrategy method, that may be called to add a strategy which will remain in effect for every subsequent call to the instantiate method (unless a corresponding call to the removeStrategy method has been made).

The other option is to supply the strategy directly as part of the instantiation statement, by using the strategy pseudo-keyword. In the example below, MyStrat will be applied to the instantiation of T and U, and will then be discarded (and subsequently garbage collected):

```
PT.instantiate {
   strategy MyStrat {
      template T
      template U
} }
```

A slight variation of the above is to specify a strategy only for a part of an instantiation. In the example below, the strategy will only apply to the U template. Thus, anything that involves T, such as a merge of classes from these two templates, will not be affected by the strategy.

```
PT.instantiate {
  template T
  strategy MyOtherStrat {
     template U
} }
```

Specifying a strategy in this way also has the added implication that only interception points A, B and C from Figure 2 are valid for strategies that do not cover an entire instantiation. This is due to the fact that any later interception points will potentially involve data from all templates specified in the instantiation (and the strategy should thus have been at the outermost level if access to such data is required).

If more than one strategy is listed in the call to the instantiate method, they are by default ordered according to the following rules: first, each interface pertains to a specific interception point, and the order in which these points are encountered is, as shown in Figure 2, from A to E. Secondly, strategies are executed in the same order as they are listed in the instantiation specification, depth first.

For local strategies, there is also the issue of *propagation* (which would be an issue for instance when code in a template instantiates other templates). With focus on aspect deployment, Tanter has treated the issue of deployment scope, and identifies three dimensions: propagation down the call stack, propagation with regards to delayed evaluation (execution of methods on objects created within a scope, after the scope has been exited) and deployment-specific join-point filtering [8]. While the latter has little relevance to us (since join-points are specific to AOP), the first two are clearly relevant. In this work, we settle for the base case of no propagation, neither down the call stack nor to delayed evaluations, and explicitly leave investigation of the other variants for future work. This means that templates instantiated within templates in an instantiation in which a local strategy is in effect, will not be affected by said strategy. That is, if a given template `A` (or a class within it) instantiates a template `B`, and template `A` is instantiated with a local strategy `S`, `S` will *not* apply to the instantiation of `B`. However, a local scope that propagates down the call stack can be emulated by adding a global strategy at the first interception point, and subsequently removing it again at the last.

By utilizing the different options for specifying strategies, it is possible to both apply different composition semantics to a controlled portion of the code, and also to mix different semantics in one program, by utilizing different strategies. Furthermore, one can base strategy selection upon runtime conditions (e.g. such as whether a debug or trace flag has been set).

3.4 Alternate Semantics for Conflict Resolution

When performing a merge of previously independent (template) classes to form a new class, several conflicts may occur. The basic strategy employed by GPT (and, indeed, the static version of PT) is to require an explicit resolution from the programmer. However, it is not at all obvious that any given strategy is preferable for every case. For instance, when a conflict due to duplicate method definitions occur, there are at least a few reasonable approaches to take. The framework could view methods belonging to the class listed first in the call to `instantiate` as having precedence, and discard any conflicting methods stemming from classes appearing later in the instantiation statement (this, i.e. using order to resolve conflicts, is, for instance the approach taken when resolving members originating from mixin inheritance in the JavaFX language [10]). Another approach would be to somehow compose the two (or more) methods, so that the corresponding method in the new class would yield the effect of calling one of the original methods after the other. The latter approach (or variants thereof) could be relevant if trying to achieve similar effects (albeit on much a smaller scale) as is attainable with composition mechanisms such as those available in e.g. HyperJ [27].

To override GPT's default method conflict resolution strategy (which is to throw an exception unless an explicit resolve is present), and provide a strategy similar to that of JavaFX mixins, we could employ the following implementation of `PTConflictResolution-Strategy`:

```
class OrderSignificantMergeStrat implements
   PTConflictResolutionStrategy {

   def onMethodConflict(classNode1, methodNode1,
      classNode2, methodNode2, instantiation) {

      // remove the method from the class from the
      // last of the two corresponding templates
      classNode2.getMethods().remove(methodNode2)
   }

   // corresponding implementation for field conflicts
   // here, and (possibly) empty implementations of
   // the remaining methods of PTConflictResolution-
   // Strategy
}
```

To utilize this particular strategy when performing an instantiation, we could for instance supply it explicitly to the `PT.instantiate` method by utilizing a `strategy` pseudo-keyword, or utilize the static `addStrategy` method of the `PT` class, as discussed in Section 3.3. An example of the former approach is shown below:

```
PT.instantiate {
   strategy OrderSignificantMergeStrat {
      template U
      template T
} }
```

Given the instantiation above, any conflicting method signatures in merged classes would be resolved by preferring those stemming from the template `U` over those stemming from `T`. Clearly, variants of this strategy (e.g. renaming of conflicting methods) could easily be created by making slight modifications to the strategy above.

Yet another approach would be to disregard method conflicts entirely. Given a situation in which only a (well-known) subset of the templates' functionality is of interest to the user, it might be reasonable to disregard conflicts for methods that will never be called at runtime anyway. Providing the following strategy would enable such an approach:

```
class IgnoreConflictsStrat implements
   PTConflictResolutionStrategy {

   def onMethodConflict(classNode1, methodNode1,
      classNode2, methodNode2, instantiation) {
      // remove one of the methods so that the code
      // will compile:
```

```
      classNode.getMethods().remove(methodNode2)
      // replace the code of the other method
      // with just an exception:
      methodNode1.setCode(
         { () -> throw new Exception(...) })
   }
   // implementation of the rest here...
}
```

Such an approach as the one above clearly requires good unit test coverage in order to prevent exceptions from popping up at unexpected places at runtime. However, given such coverage (which doesn't seem too far-fetched in the context of a system developed in a dynamic language to begin with) it could simplify merging and template usage in general.

3.5 Extending the Instantiation DSL

At interception point A in Figure 2, there is a check for *setup strategies*. Such strategies may be applied in order to customize the way an instantiation tree representation is built from a specification, or to customize other parts of the GPT framework. We shall, as an example, look at how we can utilize this in order to provide a richer DSL for setting up instantiation specifications. The GPT framework will at interception point A look for classes that implement the PTSetupStrategy interface:

```
interface PTSetupStrategy {
   // called once upon initialization of the
   // instantiation process:
   def initialize();

   // called once to acquire the factory to
   // use when creating objects:
   def getFactory();
   ...
}
```

The getFactory method above utilizes the *abstract factory* [11] design pattern in order to return a factory; this factory implements an interface named PTClass-Factory for creating objects of the classes used by the PT class of the GPT framework. The framework contains a default implementation (that can return objects of the classes from Figure 1), but through the use of strategies, the user might change which factory will be used, and thus which classes are used for nearly every part of the instantiation process.[4]

3.5.1 A Simple AOP-like Extension

As an example extension, we shall consider adding simple AOP-like functionality that allows for specifying a set of classes with which functionality from one class should be merged. We shall utilize this to implement the typical AOP idiom of adding logging before or after method invocations to every class in a given subset of a program; here this subset will be some of the classes from a template. One option would be to specify each class in this subset explicitly, however, for non-trivial templates this would be both tedious and error prone. Thus, a better option would be if the framework could support more complex expressions for selecting classes for merge in the instantiation specification, e.g. that classes having names containing a given string or classes containing certain method signatures should be included. We will shortly get back to how we can express that, but first we will look at implementing the logging functionality.

For logging method calls, we will utilize a simple template Logging, with a single class named Logger. This class will use the Groovy MOP (utilizing the GroovyInterceptable interface) to intercept each method call, and then log the invocation:

```
template_def Logging {
   class Logger implements GroovyInterceptable {
      def invokeMethod(name, args) {
         this.&log(name, args) // log the call
         metaClass.invokeMethod(name, args) // proceed
      }
      // implementation of log method here...
} }
```

(The ampersand in the call to the log method bypasses the MOP, in order to prevent infinite recursion.)

In order to use the Logging template to log method invocations for several classes, one possible syntax for this extension could be as follows:

```
PT.instantiate {
   strategy PTMapExtensionsStrategy {
      template ToBeLogged
      template Logging {
         mapOntoEach Logger, ToBeLogged,
           { c -> c.getName().contains("...") }
} } }
```

The mapOntoEach method shown above does not exist in the framework yet, but the desired semantics are as follows: the first parameter is the name of the class that we want to map to each of the classes as specified by the second and third parameter. The second parameter is the name of the template from which to get these classes, and the third parameter is a predicate in the form a closure that takes a ClassNode (from Figure 1) and returns a bool. The method should create a new mapping in the specification from, in the example above, Logger to every class in the ToBeLogged template that satisfies the predicate (which in the concrete example above is that the name of the class should contain a specified string, represented here by an ellipsis).

[4] And thanks to the dynamic nature of Groovy, the new classes of which the factory returns objects need not have any relation to the existing ones other than in their ability to answer to the same method calls, further substantiating the flexibility of this approach.

In order to implement support for this functionality, we may employ the following implementation of PTSetupStrategy, in which we replace the default class factory with our own custom version:

```
class PTMapExtensionsStrategy
      implements PTSetupStrategy {
   def setupFactory() {
      return new ExtensionsFactory()
   }
  // implement rest of the methods of
  // PTSetupStrategy here...
}
```

The ExtensionsFactory class is a trivial implementation of the PTClassFactory, where the only significant method (for this use case) is the implementation of the createMapStatementsNode, which should return a subclass of the MapStatements class shown in Figure 1:

```
class ExtensionsFactory implements PTClassFactory {
   def createMapStatementsNode() {
      return new MapAllExtension()
   }
   // rest of the PTClassFactory methods here...
}
```

The MapAllExtension class shown in the following contains the gist of the new functionality, by defining the mapOntoEach method that is used in the instantiation above. The execution of the call in the instantiation specification is delayed until the closure in question is run, and method calls are then delegated to the MapAllExtension class.

Implementing the required functionality can be done with relative ease, by utilizing the fact that a MapStatements object has access to the instantiation, and from there one can easily get hold of the class names in a given template, as indicated by the getClasses method of Figure 1. The template is accessed through a collection (templates) on the instantiation that is indexed by the templates' name, and the map method utilized within the each loop[5] to perform the actual mappings is directly available from the MapStatements class.

```
class MapAllExtension extends MapStatements {
   def mapOntoEach(String classToMap,
         String templateName, Closure predicate) {

      instantiation.templates[templateName].
         getClasses().each {
           if(predicate(it)) {
              map(classToMap, it.getName())
} } } }
```

[5] The variable it utilized within the loop is supplied by Groovy as an automatic reference to current object from the collection over which the loop is being performed. Thus, in the example it will be a reference to an object of the ClassNode class.

Clearly, more elaborate schemes for selecting classes than just enumerating every single one within a template an applying a boolean condition can be created. Still, this simple example, in our opinion, demonstrates one important aspect of the flexibility of our framework, i.e. the ability to easily extend not just the semantics of existing functionality, but also to add whole new concepts to the mix.

3.6 Meta-Aware Package Templates

The example strategies presented above have all been specified as part of an instantiation specification in the class that calls the PT.instantiate method. However, part of the motivation behind the original package template mechanism, and indeed this work as well, has been to facilitate the creation of self-contained, reusable entities that typically span multiple classes. From that point of view, there is another interesting possibility: to take advantage of the fact that all the power of our meta-level API can be made available to the package templates themselves, not only when they instantiate other package templates, but also as a way to hook onto their own instantiation process. This entails that templates in GPT may be introspective with respect to their own instantiation, and e.g. specify instantiation behavior, constraints or requirements that are inherent to the templates themselves, relating for instance to the target classes of a merge.

To utilize a strategy within a template, the template needs to contain an implementation of one or more of the strategy interfaces defined in the text preceding this section. In addition to implementing the interface, an annotation @PTMeta is required, to enable the GPT framework to cleanly separate meta-level template classes from base-level ones.

Meta-level template classes may as such not be explicitly referred to as part of an instantiation specification, and will thus not be part of the resulting package. Strategies internal to a template are at instantiation time applied prior to strategies (implementing the same interface) specified explicitly in the call to instantiate.

3.6.1 A Reusable Implementation of the Active Object Pattern

We have in previous work [3] discussed the application of package templates to provide reusable and pluggable implementations of some design patterns from [11]. Here, we consider how this can be done for another pattern that to begin with does not lend itself very well to pluggability, by taking advantage of meta-level strategies within templates.

The *Active Object* pattern is a design pattern for concurrent applications, and is described in detail in [20]. The pattern revolves around four roles: the *ser-*

vant or *active object* performs operations at the request of *clients*. A proxy sits between these two, and routes requests (i.e. method calls) through a *scheduler*, which in turn schedules and, subsequently, starts operations on the servant. The proxy immediately returns a *future* [21] to the client in place of the actual return value of the operation. The future contains methods for checking for the completion of the operation, and a (potentially) blocking method to wait until the result is ready.

Given the inherent relative complexity of this pattern, having a reusable, pluggable, implementation would supposedly be a major convenience. Below we show a skeleton of the involved classes as a package template, with pseudo-code enclosed in angled brackets:

```
template_def ActiveObjectPattern {
  class Scheduler {
    def queue = < a fitting queue abstraction >
    synchronized def enqueue(operationName, args) {
      def future = new Future()
      queue.put(operationName, args, future)
      < start scheduler thread if not started >
      return future
    }
    < actual scheduling code here >
  }

  class Future {
    private def value
    private def completed = false
    synchronized def getValue() { ... }
    synchronized def setValue(val) { ... }
    def isCompleted() { return completed }
  }

  class ActiveObject { /* see below */ }
  class ActiveObjectProxy { /* see below */ }
}
```

For optimal ease of use and conceptual simplicity for the developer, it would be desirable if one could just merge the `ActiveObject` class with an appropriate domain class, and then proceed to use the latter regardless of its newly added active behavior. The syntax that we would like to have is something resembling the following instantiation and subsequent usage:

```
def INST_H = PT.instantiate {
   template HeavyLifting // contains ComplexProcess
   template ActiveObjectPattern {
      map ActiveObject, ComplexProcess
} }

def c = new INST_H.ComplexProcess()
def result = c.PerformLongRunningTask()
// do something else for a while...
// then (potentially) block and wait for the result
println result.getValue()
```

In order to achieve this, we would need a transparent proxy to handle the wrapping of method calls. Luckily, such proxies are relatively easy to create in most languages with proper support for a MOP, including Groovy. Hence, we may implement the `ActiveObjectProxy` as follows:

```
class ActiveObjectProxy implements
      GroovyInterceptable {
   static def scheduler = new Scheduler()
   def invokeMethod(name, args) {
      return scheduler.enqueue(name, args)
} }
```

This proxy simply intercepts every method call (as implied by the implementation of the `GroovyInterceptable` interface), and translates each call into a request that is enqueued on the scheduler. The scheduler will, as previously shown above, immediately return a future.

The active object itself, represented by the `ActiveObject` class in the template, is really only a placeholder for the functionality provided by the domain class. As such, we may actually leave this class completely empty in the template definition. However, since we are interested in *substituting* (from the user's point of view) the domain class with the proxy, and then subsequently routing calls to it through the scheduler, we need to employ a strategy to handle this. Since this strategy is a part of the pattern implementation itself, it makes sense to keep it within the `ActiveObjectPattern` template. The strategy is actually quite simple, and its only goal will be to remap the active object and proxy template classes, respectively.

```
template_def ActiveObjectPattern {
   @PTMeta
   class MappingStrategy
         implements PTInstSpecStrategy {
      def processInstantiationSpec(inst) {
         def aoName = "ActiveObject"
         def proxyName = "ActibeObjectProxy"
         def implName = "ActiveObjectImpl"

         inst.setMappedName(aoName,
            inst.getMappedName(aoName))

         inst.setMappedName(inst.
            getMappedName(aoName), implName)

         inst.setMappedName(aoName, implName)
   } }
   < rest of the pattern classes here >
}
```

By supplying the strategy above along with the classes representing the actual roles of the template, we allow the template to capture and encapsulate the conceptual intent of the pattern, which is to substitute a synchronous long-running task with an asynchronous approach through queueing requests. The user may express his or her intent by mapping the active object to an appropriate domain class (representing the time-consuming task), and not worry about how the proxy works (or even caring that there indeed is one).

3.7 Interactions

Strategies might (obviously) interact with one another, and this might be deemed unfortunate. For instance, a strategy specified as part of an instantiation specification might interact with a strategy internal to the template(s) being instantiated, without the developer of the former even being aware of the latter (thus, what might be considered an advantage from an encapsulation and ease-of-use point of view might be seen as a hinderance or inconvenience in situations with multiple strategies).

Since strategies are developed in imperative code (and that imperative code might be highly dynamic and type-less), there is no good way to automatically determine interactions up-front. Thus, the strategy developer needs to take special care to document the workings and assumptions of the strategy, and the subsequent user needs to order strategies according to the desired results. If only one strategy (or a specific set of strategies) is required, an `exclusive` modifier may be applied to the `strategy` keyword in an instantiation, e.g. as shown below:

```
PT.instantiate {
  exclusive { strategy S {
     template T
  } } }
```

This would prevent any other strategies than `S` from executing.[6] However, the `exclusive` clause is merely a small stroke on a large canvas of ways to deal with interactions, and this is clearly an area in which more research could be fruitful.

4. Discussion

Even though the meta-level functionality presented in this paper is built on top of the basic GPT mechanism as presented in [2], which is an implementation of the package template mechanism specifically targeted at the Groovy language, it is our opinion that the mechanism has a great deal of conceptual generality, making it applicable to other composition mechanisms and other programming languages besides Groovy.

We observe that a composition in general relies on a specification of what is to be composed and how, and for GPT this corresponds to the instantiation specification. Other examples include Groovy's own runtime mixin mechanism, for which the specification is the classes mentioned as arguments to the `mixin` method, and the Traits mechanism [28], for which the specification would be the set of traits and potential renames and exclusions specified in the trait list of a class.

Furthermore, we suggest that a representation of such composition specifications can be made available for modification by implementations of meta-level interfaces that are tailored to the composition mechanism at hand, and that such implementations can subsequently be invoked throughout the runtime composition process in order to control it. Applying this to e.g. the Traits mechanism, the meta-level interfaces could offer methods to be called for trait composition, for method exclusions and for conflict resolution, in much the same way that our framework does for template class composition. The pre and post instantiation/trait composition interception points could also still apply.

However, the overall applicability of our approach is hard to properly assess, both in terms of generality with regards to different composition mechanisms and in terms of usability for the programmer, without having applied it to a larger set of real-world scenarios. Thus this, necessarily paired with lifting the implementation from a proof-of-concept level, are important directions for future work. Nevertheless, the examples presented herein can at least serve to give us an idea of the potential usability of this work, and as such motivate a more thorough investigation.

5. Implementation

As mentioned earlier, the meta-level mechanisms of this work is built on top of work that we described in [2]. In that work, the basic GPT mechanism (or, in the terminology of [16], the "primary interface"), without the meta-level mechanisms presented here, was developed. By keeping, to a large extent, the previous implementation of basic GPT, many of the design choices made back then also applied this time around, e.g. to only make use of the standard features of Groovy itself, and not modify any of its source code, and to implement the functionality as a Java archive (jar) that can be included in any Groovy program in order to utilize GPT. One of the things that we touched on in [2] in that respect was the trade-off between a clean syntax (like the one in standard PT [19], that requires a dedicated parser), and ease of implementation by utilizing a tiny internal DSL to express instantiations. With this work, it seems clear that keeping everything in standard Groovy has paid off, since the addition of meta-level features could be done in a relatively straight-

[6] The `exclusive` pseudo-keyword could with relative ease be implemented by a global strategy in the manner described in Section 3.5 (carefully making sure of not including itself in the exclusivity, obviously).

forward and open manner. However, were it only for the ease of implementation it would perhaps not be all that significant, but the choice to keep the entire mechanism within the realms of Groovy means that it also enables the user to extend and modify the framework with relative ease. In other words, the implementation backs the desiderata of supporting *unanticipated* extension and modification, as seen for instance in Section 3.5.

The implementation utilizes Groovy AST transformations to transform occurrences of `new` statements that refer to template classes into reflective runtime calls. An extended Groovy parser is utilized to transform templates into an AST representation that can subsequently be manipulated by user implementations of the meta-level interfaces. The tiny internal DSL for instantiations is implemented by utilizing Groovy closures for which the execution is delayed until they are assigned a specific delegate that overrides the `methodMissing` and `propertyMissing` meta-object protocol methods. Applying this in a builder-like fashion (resembling e.g. the Groovy `MarkupBuilder`), the framework dynamically builds a tree representing the requested instantiation at runtime, including any elements added to the DSL by strategies as exemplified in Section 3.5.

Performance While performance tuning has not been a main activity in our work with GPT thus far, and our testing in this area is far from comprehensive, it seems reasonable to include a few words on the topic. All tests were run on a MacBook Pro with a 2.53 GHz Intel Core 2 Duo processor and 4 GB RAM. The operating system was Mac OS X 10.6.4, and all the tests were run from a standalone Groovy script file in the NetBeans IDE [24] version 6.7.1, with Groovy version 1.7.1 and JVM version 1.6.0_20.

Object creation from template classes, i.e. statements of the form `new INST_T.A()`, are in the order of 2 to 3 times slower than their plain Groovy counterparts. This is not surprising, since our approach is based on reflective `newInstance()` calls, while Groovy itself resolves plain `new` statements a compile-time.

Method call, method execution and field access to objects created this way incur no performance hits compared ordinary Groovy objects.

Template instantiations have no direct counterpart in Groovy with which a reasonable comparison can be made, since ordinary usage of the `import` statement is handled at compile time. However, we can at least give a few indications of the performance implications of runtime instantiations. On our test setup, the runtime instantiation of a single simple (containing just a few classes) template took in the order of 30 to 40 ms. This includes both the parsing and modification done by GPT, and the application of the Groovy compiler itself. However, this time will increase in a linear fashion with the number of classes, and for complex templates the instantiation time might quickly become a concern for real applications. However, it should be noted that no attempts at optimizing the code for instantiations have been performed at present.

When meta-level strategies are involved, an important factor will of course be what the actual code in these strategies does. However, utilizing rather simple strategies, e.g. such as the one in the Active Object example from Section 3.6.1, during instantiation does not introduce any overhead that we have been able to reliably measure.

6. Related Work

Virtual classes were introduced in the BETA language [22, 23]), and have subsequently been utilized in a number of mechanisms, such as e.g. gbeta [9], J& [25] and Caesar [1] (the latter also contains AOP features, see below). The main idea is that an outer class encloses one or more inner classes that are virtual in the sense that they can be overridden by defining new versions in subclasses of the outer class. This provides some of the same flexibility for unanticipated adaption as GPT; a main difference, however, is that in GPT this is all handled during instantiation, and there need not be any sub/superclass relationship between adaptor and adaptee. J& and gbeta support multiple inheritance, which is something that GPT does not allow. However, for code composition, we can attain some of the benefits of multiple inheritance by merging template classes. In NewSpeak [4], the *only* modularity construct is the class, and nesting is applied to provide a module hierarchy. All class references are virtual, and thus every class can function as a mixin. Like GPT, NewSpeak allows for multiple independent instantiations of the same module. A module can be adapted and extended by creating subclasses of outer and inner classes, and overriding virtual definitions. However, since GPT is based on templates whose classes become ordinary classes only after an instantiation, the range of allowable modifications can be greater than in typical virtual class-based mechanisms.

Subject-Oriented Programming (SOP) [15] is an approach that allows multiple subjects to have differing, possibly incomplete, views of objects in a system. A subject represents a view of the world, i.e. a subjective perspective, as manifested by state and available behavior. The stated primary goal of the mechanism is to *"facilitate the development and evolution of suites of cooperating applications."* [15, page 412]. Multiple subjects can be composed to allow them to interact, and to react to each other's behavior. Subject composition is governed by composition rules that may be general or specific, though [15] explicitly leaves the formalization of such rules for later work.

In [26], a model for SOP composition rules is described, relying on *labels* that describe the declarations in a (flattened) subject. Composition clauses are then defined for such labels, which again may be used to define more high-level composition rules.

GPT classes, and thus package templates, are declarationally complete, and do not, as opposed to SOP subjects, *inherently* represent a subjective and incomplete view of the global system. Rather, templates are to be thought of as complete units in their own right, that *may* be composed with other units to form a larger whole. SOP does not define a model for runtime (or, for that matter, compile-time) interaction with composition rules.

Aspect-oriented programming (AOP) [18] can be seen as a special case of meta-level programming in the sense that meta-level entities (the aspects) make changes to base-level code, and the base-level is *oblivious* to such changes. However, the changes that an aspect is allowed to perform are typically quite limited compared to the meta-level of GPT. For instance is renaming of classes or methods typically not allowed, nor is class merging. On the other hand, aspects have the ability to impact an entire application (they are typically global in scope), while the applications of GPT's meta-level are typically quite narrow.

In [8], Tanter describes a mechanism for controlling the scope of dynamically deployed aspects (including propagation down the call stack and to new objects), however, these aspects may, as opposed to runtime instantiations in GPT, affect behavior only, and not class structure or hierarchy.

With a Meta-Aspect Protocol [6], the semantics of an aspect-oriented language is in the hand of the programmer, allowing for control over e.g. advice ordering, aspect interactions etc. While the subject matter is somewhat different (i.e. a meta-level protocol for AOP instead of module composition), the concrete approach in that paper, which also includes an open implementation in Groovy, was indeed an important inspiration for our work.

The **Groovy** language itself contains a powerful feature in its support for compile-time **AST transformations**, and we utilize this in our implementation to a certain extent in order to transform statements relying on classes generated at runtime by an `instantiate` method call. The implementation of the meta-level API presented herein also relies on transforming (template) ASTs, however, the built in AST transformations in Groovy are not flexible enough on their own to provide the level of customizability that we need in our framework. In particular, the different phases supported by our interception points would be very hard (if at all possible) to implement in that way. Furthermore, our framework provides a much higher degree of abstraction, and supplies meaningful meta-level constructs as opposed to just directly providing the means to manipulate the AST.

7. Concluding Remarks and Future Work

We have shown how module composition mechanisms, as exemplified by the GPT mechanism, can support varying composition semantics through the use of meta-level strategies that exploit an API that gives access to composition/instantiation specifications and provides hooks at specified points throughout the instantiation process. Strategies can be applied either globally, or within explicitly defined scopes. This further entails that a program can contain a variety of different composition semantics that can even be changed throughout the lifetime of a program, based on runtime conditions.

We have also presented an approach that allows meta-level code to be encapsulated within a module, enabling the module to take control of its own instantiation. The API through which meta-level functionality is accessed is easily extensible, allowing the user to e.g. add new keywords for new functionality in instantiation specifications.

Directions for future work include creating a more robust and performant implementation, and to carry out real-world experiments with the mechanism. Furthermore, we would like to explore meta-level strategies also for the statically typed version of PT as well as for the dynamic version. Work is in progress with respect to supporting statically typed runtime instantiations [29], and strategies (both runtime and compile-time) seem to hold interesting possibilities.

8. Acknowledgements

The work reported in this paper has been done within the context of the SWAT project (The Research Council of Norway, grant no. 167172/V30). The authors would like to thank Fredrik Sørensen for many interesting and insightful discussions pertaining to this and related topics, and Vebjørn W. Axelsen for valuable feedback on the manuscript. Finally, we would like to thank the anonymous reviewers for many suggestions on how to improve the content and presentation of this paper.

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. Axelsen and S. Krogdahl. Groovy package templates - supporting reuse and runtime adaption of class hierarchies. In *Proc. Dynamic Languages Symposium '09*, 2009.

[3] E. Axelsen and S. Krogdahl. Towards pluggable design patterns utilizing package templates. In *Proc. Norsk Informatikkonferanse '09*, 2009.

[4] G. Bracha, P. V. D. Ahé, V. Bykov, Y. Kashai, and E. Mir. Modules as objects in newspeak. In *ECOOP 2010: Proc. 24th European Conference on Object Oriented Programming*, 2010.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proc. Conf. O-O. Prog.: Syst., Lang., and Appl. / Proc. ECOOP*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[6] T. Dinkelaker, M. Mezini, and C. Bockisch. The art of the meta-aspect protocol. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2009. ACM.

[7] R. Eckstein. Mixins in JavaFX 1.2 technology. Sun Microsystems, URL: `http://java.sun.com/developer/technicalArticles/javafx/mixin/`, 2009.

[8] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 168–179, New York, 2008. ACM.

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

[10] R. Field. JavaFX language reference. Sun Microsystems, URL: `https://openjfx.dev.java.net/langref/`.

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

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[13] The Groovy homepage. URL: `http://groovy.codehaus.org`.

[14] Groovy: Runtime vs compile time, static vs dynamic. URL: `http://groovy.codehaus.org/Runtime+vs+Compile+time,+Static+vs+Dynamic`.

[15] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428, New York, NY, USA, 1993. ACM.

[16] G. Kiczales. Beyond the black box: Open implementation. *IEEE Softw.*, 13:8–11, January 1996.

[17] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[18] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[19] 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.

[20] G. R. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc.Pattern Languages of Programs,*, 1995.

[21] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM.

[22] O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM.

[23] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[24] The NetBeans homepage. URL: `http://netbeans.org/`.

[25] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06*, pages 21–36, New York, 2006. ACM.

[26] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. *SIGPLAN Not.*, 30(10):235–250, 1995.

[27] H. Ossher and P. Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 734–737, New York, NY, USA, 2000. ACM.

[28] 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.

[29] F. Sørensen, E. W. Axelsen, and S. Krogdahl. Dynamic composition with package templates. In P. Lahire, editor, *Proceedings of the First International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines*. CEUR-WS.org, 2010.