

Reflecting on an Existing Programming Language

Andreas Leitner, Dept. of Computer Science, ETH Zurich, Switzerland
Patrick Eugster, Dept. of Computer Science, Purdue University, USA
Manuel Oriol, Dept. of Computer Science, ETH Zurich, Switzerland
Ilinca Ciupa, Dept. of Computer Science, ETH Zurich, Switzerland

Reflection has proven to be a valuable asset for programming languages, especially object-oriented ones, by promoting adaptability and extensibility of programs. With more and more applications exceeding the boundary of a single address space, reflection comes in very handy for instance for performing conformance tests dynamically.

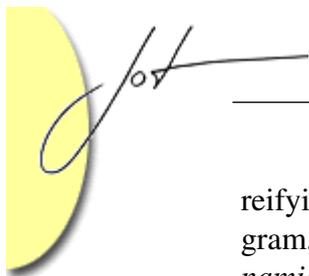
The precise incentives and thus mechanisms for reflection vary strongly between its incarnations, depending on the semantics of the considered programming language, the architecture of its runtime environment, safety and security concerns, etc. Another strongly influential factor is legacy, i.e., the point in time at which the corresponding reflection mechanisms are added to the language. This parameter tends to dictate the feasible breadth of the features offered by an implementation of reflection.

This paper describes a pragmatic approach to reflection, consisting in adding introspection to an existing object-oriented programming language *a posteriori*, i.e., without extending the programming language, compiler, or runtime environment in any specific manner. The approach consists in a pre-compiler generating specific code for types which are to be made introspectable, and an API through which this code is accessed.

We present two variants of our approach, namely a homogeneous approach (for type systems with a universal root type) and a heterogeneous approach (without universal root type), and the corresponding generators. We discuss limitations such as infinite recursion, and compare the code generated with these two approaches by Erl-G, a reflection library generator for the Eiffel programming language, thereby quantifying the benefits of a universal root type. Erl-G is being used by several tools for development in Eiffel.

1 INTRODUCTION

Reflection [15] has made its way into many programming languages in one form or another. By enabling the reification of the structure of a program and its abstract data types, *structural* reflection enables the program to reason about the very data it manipulates. *Introspection* is a subset of structural reflection focusing mainly on representing the runtime structure of a program without supporting its alteration. *Behavioral* reflection consists in



reifying the very programming language semantics and the data used to execute a program, thus providing the possibility to manipulate operational semantics at runtime. *Dynamic* invocations are a subset of behavioral reflection that allows to invoke code based on the results given by structural reflection.

In the realm of object-oriented programming languages, reflection has become especially popular, as it further supports adaptability and extensibility of programs, two of the inherent driving forces behind the object paradigm. With many distinguished characteristics of a given application translating to the creation of differential object types and subtypes, and more and more applications exceeding the boundary of a single address space, introspection comes in very handy for performing conformance tests dynamically, and hence for mediating between different type domains. Furthermore, introspection and dynamic invocations allow a program to adapt its behavior dynamically, depending on its internal state or on external stimuli, thus circumventing some of the limits present in statically typed languages.

The incentives and thus mechanisms for reflection vary strongly between its incarnations, depending on the semantics of the considered programming language, the architecture of the runtime environment, safety and security concerns, etc. Another strongly influential factor is legacy, i.e., the point at which the corresponding reflection mechanisms are added to the language. In the dynamically typed Smalltalk language [10], (structural) reflection appeared from the start as an integral part of the computation model, going as deep as affecting the very language foundations — more precisely the message dispatching mechanism — by providing each object with a “default method” (*doesNotUnderstand*) executed in the absence of a more specific method able of handling a given invocation. In Java [11], introspection was added gradually with hooks into the virtual machine and only a limited flavor of behavioral reflection eventually appeared, without any support from the runtime environment, in the form of *dynamic proxies* [25]. In the AspectJ [16] extension of Java, the interface to behavioral reflection mechanisms (the “meta-object-protocol”) has even made its way into the language syntax alike in 3-Lisp (level-shifting processor) [8]. The new viewpoint offered thereby on program design and development has even culminated in the coining of a term denoting an intriguing new development methodology — “aspect-oriented software development”.

In this paper we describe a more pragmatic approach to reflection, consisting in adding introspection to an existing object-oriented programming language *a posteriori* and *without extending* the programming language, compiler, or runtime environment in any specific manner. To that end, we introduce (1) a concise API for programmers to use, and (2) a pre-compiler which generates code for the introspectable types. This approach yields the clear benefit of dealing inherently with the legacy factor, and supporting the use of different compilers and runtime environments. We illustrate this approach through Erl-G [17], a pre-compiler generating introspection code for the Eiffel language [21]. Broadly speaking, Erl-G generates meta-classes akin to those found in Smalltalk’71 [23] for Eiffel classes. Erl-G makes it possible to introspect programs written in the Eiffel programming language, spanning many of the language constructs in a simple and uniform way, regardless of the compiler (ISE, SmartEiffel, etc.) or runtime environment (standard, .NET, etc.)



at hand.

We first introduce an abstract core language which we use to show the semantics of two reflection code generation methods. The first, *heterogeneous* approach does not require the existence of a universal root type (i.e., a type without super-type to which *all* types in the system conform), whereas the second, more efficient, *homogeneous* approach does rely on the presence of such a type. Introspection is accessible through a generic API, whereby programs become entirely detached from the introspection code itself, removing any explicit dependencies between programs using introspection and the generated code.¹ We then illustrate through the Eiffel programming language how this simple language can be mapped to a concrete one. We present the full functionalities of an implementation of our approach in Eiffel, going by the name of Erl-G.

We discuss general and Eiffel-specific intricacies and limitations, and how to mitigate these. For instance, we show how generic types can lead to infinite recursion and thus to an infinite number of meta-classes being generated in the absence of a universal root type; we also show what is the minimal support our approach requires from a programming language, and how the case of Eiffel allows for introspection over generic arguments, a lack in the recent addition of generics to Java which has made the subject of many discussions.

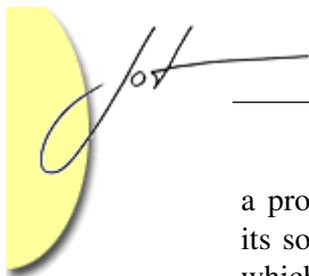
Last but not least, we show experimental results *quantifying* the benefits of a universal root type, by illustrating the significant decrease (averaging around an order of magnitude) achieved in the number of generated meta-classes and in system compilation time when introducing such a type.

Roadmap. Section 2 presents the semantics of our generation approaches based on a simple abstract language. Section 3 illustrates how our approach can be mapped to a concrete programming language, by using Eiffel as an example. Section 4 then explains how the generation approaches have been implemented in the Erl-G tool. Section 5 evaluates our implementation on several real world programs in terms of efficiency, and discusses the applicability of our approach. Section 6 discusses some of the benefits of the reflection capabilities offered by Erl-G by presenting examples of development tools that rely on it. Finally, Section 8 draws conclusions.

2 REFLECTION GENERATION

Reflection offers a program the ability to query and/or modify its state and execution, as well as change the semantics and implementation of the programming language that it is written in. The two fundamental, complementary, aspects of full reflection are apparent from this definition: *structural reflection* allows a program to inspect and modify its own state as it would do with any other data that it manipulates; *behavioral reflection* allows

¹In this sense, this generic API is similar to the interfaces of remote objects in Java RMI implemented by proxies generated by the *rmic* pre-compiler, and only required at runtime.



a program to modify its code, as well as the very semantics and execution model of its source programming language. Both these levels of reflection require a process by which execution state is represented and made available to the program as ordinary data; this process is called *reification*. While the latter type of reflection has been devoted much attention in the past (see Section 7), we focus in this paper on the former type of reflection, more widely encountered in some form or another, yet still lacking support in many languages.

Approach Overview. As mentioned, many languages have been initially devised without support for reflection. In this paper we deal with the more compelling case where a considered language is to be added reflective abilities, but should not change in any way. As a consequence, the approach we propose is based on the generation of a set of meta-classes which represent the structure and state of the program to be reflected upon. These classes embody the necessary reification structure and make up a *reflection library*. Programs access these reflective features through the use of a regular library. Its API (described in detail in Section 3) offers the most common facilities needed for structural reflection: retrieving a class by its name, creating new objects, calling a method, getting the value of an attribute, etc. Note that this API so far is not biased by any feature characteristic to any one particular programming language; rather, it allows for a set of operations that are required for structural reflection in a large family of object-oriented programming languages. The implementation of this API is then achieved through different means depending on the possibilities of the target language, and the API's precise semantics is obviously guided by the semantics of the target programming language.

A Simple Language. Figure 1 shows the syntax of a very simple abstract language on which the transformations are defined. The supported features are inspired by the Eiffel language, which is targeted by our main implementation. In particular, the proposed core language includes multiple subtyping through multiple inheritance and generics, a feature present already early in Eiffel, and which nowadays enjoys various levels of support in many mainstream languages.

$\langle V, W, \dots \rangle$ represents a declaration which involves different literals V , W , etc. The notation $[X_1, \dots, X_n]$ denotes a list made of literals X_1, \dots, X_n . The \oplus operator denotes list concatenation. Y^* denotes zero or more literals Y . Y^+ denotes one or more literals Y . For example, a class is defined by a name and optional sets of (1) generic arguments, (2) super types, (3) constructors, (4) fields, and (5) methods respectively. A type is defined by a class (its so-called *base class*, see Section 3), and actual types for all generic arguments. In other terms, a type is defined by a class, in which all generics have (recursively) been linked to types. A class with “open” generic arguments gives rise to a family of types but does not define a type by itself.

Traditionally names are considered to belong to different subsets of the names set depending on the way they are used. This differentiation is however not relevant for the scope of this paper and subsequently ignored. Note however that names can be handled as expressions, by applying the $ID()$ operator. The result can be viewed as a reification

<i>names</i>	$N ::= v \mid g \mid m \mid cl; \text{ID}(N) \in T_{NAME}$
<i>application</i>	$A ::= \langle [CL^+], \text{create } T.c(\mathbf{Void}) \rangle$
<i>class</i>	$CL ::= \langle cl, [G^*], [T^*], [C^*], [F^*], [M^*] \rangle$
<i>generic</i>	$G ::= \langle g, T \rangle \mid \langle g, \perp \rangle$
<i>attribute</i>	$F ::= \langle V, E \rangle$
<i>variable</i>	$V ::= \langle v, T \rangle \mid \langle v, g \rangle$
<i>type</i>	$T ::= \langle cl, [T^*] \rangle$
<i>constructor</i>	$C ::= \langle c, [V^*], E \rangle$
<i>method</i>	$M ::= \langle m, [V^*], T, E \rangle \mid \langle m, [V^*], \perp, E \rangle$
<i>expression</i>	$E ::= v \mid E.m([E^*]) \mid \text{ID}(N) \mid v := E \text{ in } E$ $\mid \text{if } E \text{ then } E \text{ else } E \mid \perp \mid \mathbf{Current}$ $\mid \text{create } T.c([E^*]) \mid E == E \mid \mathbf{Error}$ $\mid E.v$

Figure 1: Abstract language

of the name of some method or field, rather than its evaluation, and reflects the first-class character strings found in many object-oriented programming languages.

The actual type system is not considered in detail at this point. A class can have several super-classes, and a formal generic argument can optionally involve a bound. Any given type system adds restrictions to the generation of meta-classes outlined in the following paragraphs. The only distinction we make subsequently is between (1) a heterogeneous and (2) a homogeneous approach. In the former case, we assume that there is no global super-type to which everything conforms (akin to Ada'95, C++, Eiffel prior to becoming an ECMA standard), and thus generate a meta-class *for each type*. In the presence of a universal root type (e.g., Java modulo primitive types, Oberon, now also ECMA Eiffel), we assume the same meta-class can be used to introspect a family of types by creating a meta-class *per class to reflect* only. Ramifications of these approaches are discussed in Section 4.

Heterogeneous Approach. In the first case mentioned above a class is generated for *each type* that should be made introspectable. Figure 2 shows the semantics of the generation process $\mathcal{MT}[\cdot]$ for a given meta-class. The operator $\mathcal{MT}[\cdot]$ has a single argument, which is a type. The operator generates a class containing three methods, namely (1) *attr_val*, (2) *inv_meth*, and (3) *cre_obj*, which respectively fetch the values of a given attribute, invoke a given method, and instantiate the class through a given constructor.

Figure 3 shows how to extend the application and enable reflection for a given set of types. The principle consists in generating a new class for each base class that needs reflection and to use the relevant types in the input.

Note that in this approach a meta-class is generated for each type to be made introspectable. In the absence of classes with generic arguments, there is a one-to-one correspondence between classes and types. In the presence of generic classes, each class allows for the construction of (potentially infinitely - see Section 5) many types.

$$\begin{aligned}
\mathcal{MT}[\cdot] : T &\rightarrow CL \\
\mathcal{MT}[T_0] &= \langle mc.T_0, \emptyset, \emptyset, c_0, \emptyset, [M_{attr_val}, M_{inv_meth}, M_{cre_obj}] \rangle \\
\text{where:} \\
T_0 &= \langle CL_0, \dots \rangle; c_0 = \langle make, \emptyset, \emptyset \rangle \\
CL_0 &= \langle cl_0, [G_1 \dots G_n], [T_1 \dots T_m], [C_1 \dots C_p], [F_1 \dots F_q], [M_1 \dots M_r] \rangle \\
C_{i=1..p} &= \langle c_i, \dots \rangle; F_{i=1..q} = \langle \langle v_i, \dots \rangle, \dots \rangle; M_{i=1..r} = \langle m_i, \dots \rangle \\
M_{attr_val} &= \langle attr_val, [\langle attr_name, T_v \rangle, \langle targ, T_{ANY} \rangle], T_{VAR}, E_{attr} \rangle \\
E_{attr} &= \text{if ID}(v_1) == attr_name \text{ then } targ.v_1 \text{ else} \\
&\dots \\
&\text{if ID}(v_n) == attr_name \text{ then } targ.v_n \text{ else Error} \\
M_{inv_meth} &= \langle inv_meth, [\langle meth_name, T_{NAME} \rangle, \langle targ, T_{ANY}, \langle args, T_{LIST}[ANY] \rangle \rangle], T_{VAR}, E_{proc} \rangle \\
E_{proc} &= \text{if ID}(m_1) == meth_name \text{ then } targ.m_1(args_{1..n}) \text{ else} \\
&\dots \\
&\text{if ID}(m_m) == meth_name \text{ then } targ.m_m(args_{1..n}) \text{ else Error} \\
M_{cre_obj} &= \langle cre_obj, [\langle cons_name, T_c \rangle, \langle args, T_{LIST}[ANY] \rangle], T_{ANY}, E_{cons} \rangle \\
E_{cons} &= \text{if ID}(c_1) == cons_name \text{ then create } T_{0.c_1}(args_{1..n}) \text{ else} \\
&\dots \\
&\text{if ID}(c_k) == cons_name \text{ then create } T_{0.c_k}(args_{1..n}) \text{ else Error}
\end{aligned}$$

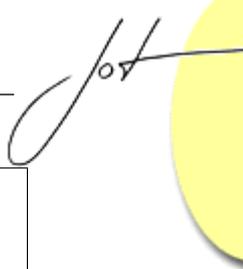
Figure 2: Semantics per type

Homogeneous Approach. In the homogeneous approach, a meta-class is generated for *each class* that should be made introspectable. Figure 4 depicts the semantics of the generation process $\mathcal{MC}[\cdot, \cdot]$. As opposed to the operator $\mathcal{MT}[\cdot]$ in the heterogeneous approach, $\mathcal{MC}[\cdot, \cdot]$ takes *two* arguments: a (base) class and a set of *derived* types for that class (a type is derived from a class iff the class is the base class of the given type). This set contains the types for which instances should be creatable via reflection.

The operator $\mathcal{MC}[\cdot, \cdot]$ generates a class containing three methods, namely (1) *attr_val*, (2) *inv_meth*, and (3) *cre_obj*. Just like in the case of $\mathcal{MT}[\cdot]$, these fetch the values of attributes, invoke methods and create instances, respectively, as shown in Figure 4. Similarly to the heterogeneous approach, Figure 5 shows how to extend an application and enable reflection.

$$\begin{aligned}
\mathcal{MT}_{app}[\cdot, \cdot] : A \times T_{LIST}[T] &\rightarrow A \\
\mathcal{MT}_{app}[A_0, [T_1, \dots, T_m]] &= \langle [CL_1, \dots, CL_n, \mathcal{MT}[T_1], \dots, \mathcal{MT}[T_m]], \text{create } T_{1.c}(\text{Void}) \rangle \\
\text{where:} \\
A_0 &= \langle [CL_1, \dots, CL_n], \text{create } T_{1.c}(\text{Void}) \rangle
\end{aligned}$$

Figure 3: Per-type reflection library generation



$$\mathcal{FL}[\cdot] : T \rightarrow T_{LIST[T_{NAME}]}$$

$$\mathcal{FL}[T_0] = [cl_0] \oplus \mathcal{FL}[T_1] \oplus \dots \oplus \mathcal{FL}[T_m]$$

where:

$$T_0 = \langle \langle cl_0, \dots \rangle, [T_1, \dots, T_m] \rangle$$

$$\mathcal{MC}[\cdot, \cdot] : CL \times T_{LIST[T]} \rightarrow CL$$

$$\mathcal{MC}[CL_0, \{T_{CR(1)}, \dots, T_{CR(t)}\}] = \langle mc.CL_0, \emptyset, \emptyset, c_0, \emptyset, [M_{attr.val}, M_{inv.meth}, M_{cre.obj}] \rangle$$

where:

$$CL_0 = \langle cl_0, [G_1 \dots G_n], [T_1 \dots T_m], [C_1 \dots C_p], [F_1 \dots F_q], [M_1 \dots M_r] \rangle$$

$$c = \langle make, \emptyset, \emptyset \rangle; T_{CR(i=1..t)} = \langle CL_0, \dots \rangle$$

$$C_{i=1..p} = \langle c_i, \dots \rangle; F_{i=1..q} = \langle \langle v_i, \dots \rangle, \dots \rangle; M_{i=1..r} = \langle m_i, \dots \rangle$$

$$M_{attr.val} = \langle attr.val, [\langle attr.name, T_v \rangle, \langle targ, T_{ANY} \rangle], T_{ANY}, E_{attr} \rangle$$

$$E_{attr} =$$

```

if ID(v1) == attr.name then targ.v1 else
...
if ID(vn) == attr.name then targ.vn else Error

```

$$M_{inv.meth} = \langle inv.meth, [\langle meth.name, T_{NAME} \rangle, \langle targ, T_{ANY} \rangle, \langle args, T_{LIST[ANY]} \rangle], T_{ANY}, E_{proc} \rangle$$

$$E_{proc} =$$

```

if ID(m1) == meth.name then targ.m1(args1..n) else
...
if ID(mm) == meth.name then targ.mm(args1..n) else Error

```

$$M_{cre.obj} = \langle cre.obj, \{ \langle cons.name, T_c \rangle, \langle args, T_{LIST[ANY]} \rangle, \langle flat.type, T_{LIST[T_{NAME}]} \rangle \}, T_{ANY}, E_{cons} \rangle$$

$$E_{cons} =$$

```

if flat.type ==  $\mathcal{FL}[T_{CR(1)}]$  then
  if ID(c1) == cons.name then create TCR(1).c1(args1..n) else
  ...
  if ID(ck) == cons.name then create TCR(1).ck(args1..n) else
  Error
...
else if flat.type ==  $\mathcal{FL}[T_{CR(t)}]$  then
  if ID(c1) == cons.name then create TCR(t).c1(args1..n) else
  ...
  if ID(ck) == cons.name then create TCR(t).ck(args1..n) else
  Error
else Error

```

Figure 4: Semantics per class

3 THE CASE OF EIFFEL

To illustrate the general solution described in the previous section, we present here its implementation in the case of the Eiffel language [19]. In a nutshell, Eiffel is purely object-oriented, strongly typed, and supports dynamic binding, (dynamic) single dispatch, and multiple inheritance. Eiffel makes for a very illustrative target language, since it innately only provides minimal support for reflection. We first overview the Eiffel language along with its type system, depicting how it impacts the implementation of the previously out-

$$\begin{aligned} \mathcal{MC}_{app}[\cdot, \cdot] : A \times T_{LIST}[T] &\rightarrow A \\ \mathcal{MC}_{app}[A_0, [T_1, \dots, T_m]] &= \\ \langle [CL_1, \dots, CL_n, \mathcal{MC}[CL_1, [T_1, \dots, T_m]/cl_1], \dots, \\ \mathcal{MC}[CL_n, [T_1, \dots, T_m]/cl_n]], \text{create } T_1.c(Void) \rangle \end{aligned}$$

where:

$$\begin{aligned} A_0 &= \langle [CL_1, \dots, CL_n], \text{create } T_1.c(Void) \rangle \\ CL_{i=1..m} &= \langle cl_i, \dots \rangle \\ [T_1, \dots, T_m]/cl_0 &= [T_{j=1..m}|T_j = \langle \langle cl_0, \dots \rangle, \dots \rangle] \end{aligned}$$

Figure 5: Per-class reflection library generation

$$\begin{aligned} \langle cl, [G^*], [T^*], C^*, [F^*], [M^*] \rangle &\hat{=} \text{class } cl[G_1, \dots, G_m] \\ &\text{inherit} \\ &\quad T_1, \dots, T_n \\ &\text{feature} \\ &\quad C_1, \dots, C_p, F_1, \dots, F_q, M_1, \dots, M_r \\ &\text{end} \\ \langle g, T \rangle &\hat{=} g \rightarrow T \\ \langle g, \perp \rangle &\hat{=} g \\ \langle v, T \rangle &\hat{=} v : T \\ \langle v, g \rangle &\hat{=} v : g \\ \langle V, E \rangle &\hat{=} V \text{ is } E \\ \langle cl, [T^*] \rangle &\hat{=} cl[T_1, \dots, T_n] \\ \langle c, [V^*], E \rangle &\hat{=} \text{create } c(V_1, \dots, V_n) \text{ is} \\ &\quad \text{do } E \text{ end} \\ \langle m, [V^*], T, E \rangle &\hat{=} m(V_1, \dots, V_n) : T \text{ is} \\ &\quad \text{do } E \text{ end} \\ \langle m, [V^*], g, E \rangle &\hat{=} m(V_1, \dots, V_n) : g \text{ is} \\ &\quad \text{do } E \text{ end} \\ \langle m, [V^*], \perp, E \rangle &\hat{=} m(V_1, \dots, V_n) \text{ is} \\ &\quad \text{do } E \text{ end} \end{aligned}$$

Figure 6: Simplified Eiffel language syntax

lined introspection mechanisms, and then look at the API of the generated Eiffel reflection library.

The Eiffel Language. Figure 6 overviews the core of the Eiffel language syntax, as a “mapping” from the abstract syntax presented in Figure 1. In the case of optional declarations (enclosed in $[\dots]$ in Figure 1), leading keywords (e.g., **inherit**) are omitted in case of absence of corresponding declarations. The same goes for separating symbols (e.g. “,”) in case of multiple declarations. Note that the *list* from Figure 1 does not appear here, as it is replaced by the generic *LIST* type.

Types. Informally, types in Eiffel are divided into three categories:

References: If an entity x is declared of type T and T is a reference type, then the value of x will be a reference to an object or *Void*.

Expanded: If an entity x is declared of type T and T is an expanded type (also called *value type*), then the value of x will be an object. The predefined expanded types (such as *INTEGER*, *REAL*, *DOUBLE*, *BOOLEAN*, and *CHARACTER*) are so-called *basic types*, but other



expanded types can also be created by using the **expanded** keyword. The basic types are implemented by library classes, but, for obvious reasons of efficiency, compilers will implement operations on them directly through machine operations, not through method calls.

Formal generics: In the case of generic classes (also called *parameterized types*), a formal generic represents a type parameter to be provided in actual uses of the class. The Eiffel genericity mechanism allows for both unconstrained and constrained (*bounded*) generic parameters; constraints in the latter case are expressed by the \rightarrow operator. An example would be the second parameter of class `HASH_TABLE [G, H \rightarrow HASHABLE]`. Any type that is a valid generic derivation of this class must have, as the second actual generic parameter, a type that conforms to `HASHABLE`.

Every type is based on a class, which is called the type's "base class". The difference between types and classes appears only in the case of generic classes: such classes do not describe a type, but a family of types. To obtain a type from a generic class, one must provide actual generic parameters. For instance, the class declared as `LIST [G]` only yields a type when a type existing in the system, such as `INTEGER`, is substituted for `G`; hence, `LIST [INTEGER]` is a type.

Subtyping. Subtyping in Eiffel is defined as follows. A type V is a subtype of a type T if and only if both:

1. The base class of V is a descendant of the base class of T .
2. If V is generically derived, its actual generic parameters must subtype those of T 's corresponding ones.

More detailed rules are described in an associated technical report [18].

Our first implementation of the meta-class generator for Eiffel was aimed at the pre-ECMA version of the Eiffel language, in which expanded types do not conform to `ANY`. As a consequence a meta-class had to be generated for every *type*, mandating the heterogeneous code generation. The introduction of a universal root type (in the ECMA standard) significantly improved the performance of the meta-class generator, since it meant that the generation of a meta-class for every type was no longer necessary: the generation of a meta-class for every *class* was sufficient. In Section 5 we assess the two approaches in more detail in terms of performance and limitations.

The Erl-G Library API. As stated in Section 2, the introspection-enabling code is generated as a library of meta-classes. One such meta-class (inheriting from `ERL_CLASS`) is generated for each class that requires introspection abilities.

The meta-model of the reflection API is represented in Figure 7. The main components of this simple meta-model are the classes `ERL_CLASS` and `ERL_UNIVERSE`. Class

ERL_SHARED_UNIVERSE implements the singleton design pattern, by ensuring that only one instance of *ERL_UNIVERSE* exists and providing the access point to it. *ERL_CLASS* is the abstract ancestor of all generated meta-classes for the classes that must be introspectable. It provides the methods for creating a new instance, retrieving the value of an attribute, and calling a method. For brevity, the arguments to these methods are omitted in Figure 7; they are however visible in Listing 1. The implementation of each method can be thought of as a big switch on the name of the constructor, attribute, or method, respectively. For languages supporting overloading the switch must be extended to also cover the types of the method arguments. Note that, since Eiffel does not support overloading, we did not need to consider this issue in our implementation.

```

class ERL_CLASS
feature
  attribute_value (name: STRING; target: ANY): ANY
  invoke_method (name: STRING; target: ANY; args: ARRAY [ANY]): ANY
  create_object (name: STRING; type: STRING; args: ARRAY [ANY]): ANY
end
class ERL_TYPE
feature
  attribute_value (name: STRING; target: VARIANT): VARIANT
  invoke_method (name: STRING; target: VARIANT; args: ARRAY [VARIANT]): VARIANT
  create_object (name: STRING; args: ARRAY [VARIANT]): VARIANT
end

```

Listing 1: Interface of *ERL_CLASS* and *ERL_TYPE*

Class *ERL_UNIVERSE* provides methods for retrieving a meta-class through the name of its corresponding class and through an object that is an instance of this class.

It should also be noted that a very limited set of features related to reflection was already present in the Eiffel language, through class *INTERNAL*. This class can check whether an object is an instance of a given type, can obtain the class name and type name for a given object, can verify type conformance, and can query and set the values of fields. The most notable capabilities missing are those of calling methods and constructors and querying the arguments of a method. However, none of the features provided by *INTERNAL* are necessary for our implementation, as explained in Section 5.

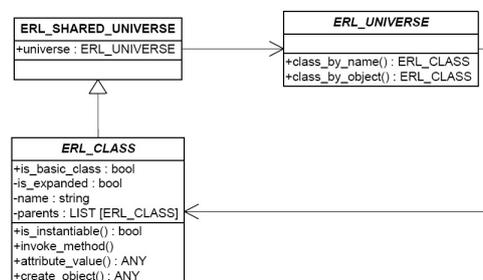


Figure 7: Meta-model for the homogeneous generation method



4 IMPLEMENTATION

This section describes how the reflection library API for Eiffel presented in the previous section can be implemented using a code generator. The general idea is to apply a pre-compiler on the code that should be made introspectable and to generate a custom-fit implementation for the generic reflection API.

Erl-G. Our generators presented above have been implemented in the Erl-G tool. More precisely, this tool implements both the homogeneous and the heterogeneous meta-class generation techniques. Erl-G is published as open source and can be downloaded as binary and source [17]. Erl-G relies on the front-end of the Gobo Eiffel parser [2] for Eiffel classes.

In the following we look at the high-level architecture of the tool, examine the details of the heterogeneous and homogeneous code generation methods, and provide an example to illustrate the approach.

General Architecture. Both methods of generating code share a common architecture. The reflection library is generated via a pre-compiler before compilation. The pre-compiler parses the system (much like the first phases of the actual compiler) and then generates the corresponding meta-classes. In scenarios where reflection is added to a platform a posteriori, it is likely that one cannot or does not want to modify the compiler itself. Our preprocessor is implemented using the Eiffel parser from the Gobo package. This parser recognizes the same language as the parser of the target compiler. The generated meta-code implements a predefined and static reflection API (described in section 3). The compiler then compiles a program consisting of the actual system, the reflection API, and the generated reflection implementation. Note that as soon as some parts of an interface within the actual system change, the pre-compilation step needs to be repeated. Figure 8 shows the relation between the various parts involved.

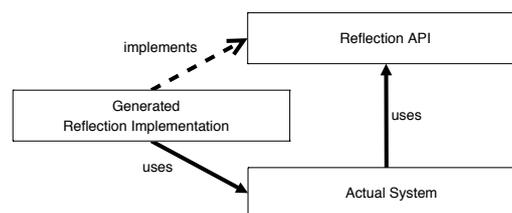


Figure 8: System with introspection support

For the pre-compiler to parse the initial system it is necessary that this system can be type-checked even before the pre-compiler has generated its meta-code. In order to solve this problem, a “dummy” reflection implementation is used. Thus, the system can be type-checked also before the real reflection implementation is available. As an added benefit, the dummy implementation allows the initial program to compile and be self-contained even if no meta-code has been generated yet.

Introspectable Types in the Heterogeneous and Homogeneous Approaches. Section 2 presented the two possibilities for generating meta-classes. Both instrumentations require a list of types (to be made introspectable) to generate the meta-classes. In practice this list is mostly generated automatically. In the heterogeneous approach, Erl-G generates a meta-class per type while the homogeneous approach has only one meta-class per class but needs the list of types instantiatable via reflection.

For the heterogeneous approach, the default behavior is to derive: (1) one type per non-generic class, (2) the most abstract derivation (i.e. the one where each formal generic parameter is replaced by its constraining type) for each generic class, and (3) each type that could be instantiated by the program at runtime. This last part is computed using the dynamic type set algorithm [20], which produces an over-approximation of the set of types that can be instantiated in the system.

For the homogeneous approach, the default behavior is to derive: (1) one type per non-generic class, (2) the most abstract derivation for each generic class. All classes are then introspectable. Non-generic classes can be instantiated and generic classes' most abstract derivations can be instantiated.

In practice, rather than redefining the default list, users can simply extend it.

Illustrations. To illustrate how Erl-G works, we use the example of a simple banking application. The main class of this application is outlined in Listing 2. Implementation details and contracts which are not germane to the subsequent presentation have been omitted for brevity. As the homogeneous approach produces less meta-code, we use it for this example.

```
class BANK_ACCOUNT
feature
  owner: PERSON
  balance: INTEGER
  make (p: PERSON; init_bal: INTEGER)
  make_with_default_balance (p: PERSON)
  withdraw (sum: INTEGER)
  set_owner (p: PERSON)
  set_balance (b: INTEGER)
end
```

Listing 2: Classes in the example

The homogeneous generation method creates a meta-class for each existing class and then one class that serves as a lookup-point for all meta-classes. Due to size restrictions we cannot show the generated code. Excerpts are published in the associated technical report [18].

Listing 3 illustrates how the reflection library generated by Erl-G can be used. Note that this code only uses the general interface described in section 3 and does not depend in any manner on the way in which the actual meta-code is generated (e.g. the static type `ERL_CLASS` is used instead of the generated descendant class).

```

class BANK_REFLECTION
inherit ERL_SHARED_UNIVERSE
feature
  execute is
  local
    c: ERL_CLASS
    p: ANY
    ba: ANY
    tmp: ANY
  do
    c := universe.class_by_name ("PERSON")
    p := c.create_object ("make", "", <<35, "John Doe">>)
    c := universe.class_by_name ("BANK_ACCOUNT")
    ba := c.create_object ("make", "", <<p, 300>>)
    tmp := c.invoke_method ("withdraw", ba, <<20>>)
  end
end

```

Listing 3: Use of reflection API

5 EVALUATION

This section evaluates the Erl-G tool and compares the two generation methods: the heterogeneous one (which generates a meta-class for every type) and the homogeneous one (which generates a meta-class for every class). First we show the results of some tests we performed using Erl-G and then discuss the applicability of the two methods.

Efficiency. We have used both the heterogeneous and the homogeneous generation methods implemented by Erl-G to make a set of libraries introspectable. As entry point for each library (in order to be able to generate an executable) we used a class implementing an interpreter. This interpreter reads instructions from the standard input and then uses the reflection API to execute the actual code (method calls, creation of objects, etc.).

Table 1 provides some figures illustrating the efficiency of both generation methods. Table 2 shows the factor gained by using the homogeneous method over the heterogeneous one. The tables show statistics of the application of Erl-G on three libraries: *Base*, the Eiffel standard I/O and data-structure library for the ISE Eiffel compiler; *Gobo*, a portable standard library and tool collection; *Vision2*, a multi-platform GUI library.

The tests were performed on a custom-built desktop PC equipped with an AMD Athlon 64 dual core and 2 GB RAM using a 64 bits version of Ubuntu Linux.

The left part of table 1 shows the results for the heterogeneous generation and the right part shows the results for the homogeneous generation. In each part the columns represent: (1) Number of classes to be made introspectable, (2) Compilation time with dummy reflection implementation, (3) Pre-compilation time, (4) Number of meta-classes generated, (5) Compilation time with generated reflection implementation. All times are given in the format hh:mm:ss.

	heterogeneous (code per type)				
	#classes	compil. wo m.-c.	gener. time	#meta-cl.	compil. w m.-c.
Base	310	00:00:24	00:01:16	1902	03:12:28
Gobo	539	00:00:23	00:05:11	2841	06:14:34
Vision2	981	00:00:21	04:40:08	9144	error (>12h)
	homogeneous (code per class)				
	#classes	compil. wo m.-c.	gener. time	#meta-cl.	compil. w m.-c.
Base	285	00:00:32	00:00:02	286	00:05:38
Gobo	530	00:00:33	00:00:03	531	00:07:54
Vision2	965	00:00:37	00:00:07	966	00:18:54

Table 1: Code generation for reflection implementations

Note, that the compilation time with the dummy reflection generation is almost constant. This is because the global analysis of the compiler determines that the executable to produce is only dependent on a fixed number of classes: the interpreter and the dummy implementation.

For the heterogeneous generation both the pre-compilation time and the full compilation time increase with the number of types in the system (which is greater than the number of classes). In the case of Vision2, the heterogeneous implementation takes over 4 hours to generate the meta-classes and the compilation of the full program fails due to overload after 8 hours. Note that the number of classes is slightly different in both cases for each library as the new version of the compiler was not backward compatible and thus it was needed to process older versions for the heterogeneous approach. In each case, the most recent version of the libraries that would still compile with the heterogeneous compiler was used.

These results show that, due to efficiency reasons, the homogeneous method should always be preferred over the heterogeneous one for languages with universally-rooted type systems. For the other languages, the homogeneous method *cannot* be used.

Applicability. As one of the primary goals of our work was to develop an approach that is applicable as widely as possible, our reflection generation method has a minimal set of requirements: the target language must provide a means of obtaining the type of an object (or, in its absence, a safe casting mechanism) and the source code for the introspectable classes must be available. A higher resolution of the type information provided by the system makes it easier to provide accurate information upon introspection. Consider the case of Java; the absence of runtime support makes it hard to provide information on



	improvement factor of homogeneous generation over heterogeneous generation				
	#classes	compil. wo m.-c.	gener. time	#meta-cl	compil. w m.-c.
Base	1.08	0.75	38	6.65	36.16
Gobo	1.01	0.69	103.66	5.35	47.41
Vision2	1.02	0.56	2401.14	9.46	∞

Table 2: Comparison of homogeneous and heterogeneous code generation implementations

actual generic arguments.

As explained in Section 4, any system contains at least as many types as classes. Hence, the heterogeneous method generates more meta-code and causes longer generation and compilation time, and a bigger memory foot-print. To compensate, a user can choose to lower the number of types that should be introspectable and Erl-G provides built-in facilities to easily customize the generation process.

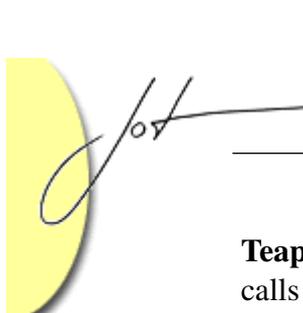
In some circumstances it is not even possible to generate meta-code for all alive types statically, since there are potentially infinitely many. Such a situation occurs for example for a generic class `SET [G]`, that includes a method that returns its powerset. The return type of such a method is `SET [SET [G]]`. This method may be used in a loop, resulting in an unknown number of types.

The heterogeneous method is limited to a fixed subset of the potentially infinite number of types. The homogeneous method does not suffer from this problem since all classes can be made introspectable, but some types will not be instantiatable through the reflection library.

The pre-compiler approach makes it possible to add reflection a posteriori to a language without the need to modify the language, the core libraries, or the execution environment. However, in the presence of dynamic class loading, pre-compilation is not sufficient. When a new class is loaded, a new meta-class needs to be generated, loaded into the program, and registered with the universe object. We did not implement such a facility in Eiffel due to its lack of support for dynamic class loading.

6 APPLICATIONS

In this section we present two applications of the Erl-G library, namely a tool for experimenting with dispatching schemes, and a framework for automatic program testing.



Teapot. A simple application of the reflection API consists in rerouting regular method calls through the reflection API. It is then possible to profile method calls, marshal results, and have, for example, a customized dynamic method dispatch mechanism.

The Teapot tool is an implementation of this idea. It currently allows to rewrite classes and redirect the calls on an instance of a given class through the Erl-G library.

Currently Teapot mostly serves as a proof of concept but its refactoring capabilities will be used further in various projects that are under development.

AutoTest. Another use of the Erl-G pre-compiler is the AutoTest tool [6], available in source and binary from [7]. AutoTest is a fully automatic testing tool for contract-equipped classes. It covers test case creation, execution, and evaluation. The user invokes AutoTest by specifying a set of classes to be tested. AutoTest then tests these classes and produces statistical results about the testing process and bug-reproducing witnesses (fragments of code which, when run, will cause a failure, thus showing the presence of the bugs).

For every class under test, AutoTest synthesizes a set of object creations and method invocations to execute. During the execution of these instructions, the contracts (in the form of preconditions, postconditions and invariants) are monitored for violations. Depending on the type of violation, AutoTest can decide whether a class was used improperly or a bug has been found.

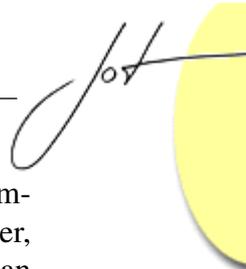
To increase robustness, AutoTest separates test case creation and execution into two separate processes. The actual execution is performed by a special purpose interpreter. This interpreter can receive (via standard input) and execute simple instructions, such as calling a method on an object, creating a new object, assigning values to variables, etc. The instructions are written in a subset of the Eiffel language.

Since this interpreter needs the ability to execute a method given by its name and the Eiffel compiler used lacks the possibility to reflect methods, Erl-G is used to implement the interpreter. Every instruction read from standard input is parsed, then the required meta-class is looked up and the reflection interface is used to invoke the requested method.

7 RELATED WORK

Recent research in the area of reflection has been strongly centered around separation of concerns and aspect-oriented programming (AOP). In the following we overview work on reflection closely related to ours, which for the most predates AOP.

Inherent Support for Reflection. Reflection has traditionally been often either at the core of a language or absent from the language. CLOS [3, 15] and Smalltalk [23] for example integrate reflection as a core concept that guides the entire program execution. In both CLOS and Smalltalk, reflection is embodied by a *meta-object protocol* (MOP).



Roughly, an MOP promotes *meta-objects* (and meta-classes) which allow the programmer to modify the behavior of the objects at runtime. To assist an MOP programmer, any single construct involved in the MOP is reified (put into an object form) and can be modified directly within the MOP. Depending on the type of modification allowed, there are different types of reflection that are available to programmers. Terminologies diverge in the literature. Generally, though, structural reflection implies a reification of the data handled by the running program; in that context introspection refers to the ability of reading that data, and intercession allows for alterations thereof. Dynamic invocation facilities circumventing (static) type checking are commonly conceived as part of introspection mechanisms. Behavioral reflection reifies the semantics of the program and the execution, and is mostly interpreted as the ability to intercept/redirect calls. Both CLOS and Smalltalk support all these kinds of reflection by design. Smalltalk for instance provides for any given object a default method *doesNotUnderstand* which is executed whenever the object receives a call for which no method with matching signature exists – a likely event given the dynamically typed nature of the Smalltalk language. Being interpreted, Smalltalk allows for strong support for behavioral reflection in the form of first class parse trees available to a program at runtime.

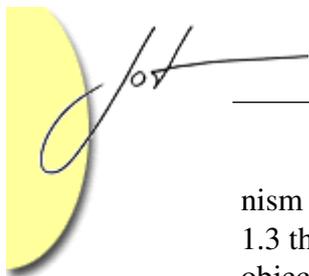
In a different setting (e.g. an existing language), it is not possible to change the core of the language and this is where our approach has added value.

Add-Ons. For languages that include only marginal support for reflection or for which no reflection support whatsoever is available, benefitting from (more advanced) reflective structures means either changing the runtime or instrumenting the code.

C++ [24] and Objective-C [22] are examples of languages devoid of any reflective features. Due to its widespread use, C++ has received particular care to accept a meta-object protocol. Efforts for providing reflection for C++ through an MOP include OpenC++ [4], Iguana [12] and MPC++ [14]. MPC++ and Open C++ claim to be meta-level, compile-time frameworks respectively, but bear more resemblances to complex preprocessing tools than to meta-object protocols, which by definition should allow run-time changes. Iguana generates the reflective infrastructure for each of given types of meta-data. While this infrastructure is the most complete one — combining introspection and behavioral reflection — it falls short in support for generics (i.e., templates in C++).

Extensions. In imperative languages like Eiffel [19], Java [1] or C# [13], reflective structures allow mainly introspection. Even within these confines, there is usually a lack of fine-grained enough features: Java does not provide a satisfactory mechanism to introspect generics, C# enables reflection at the assembly level only, and Eiffel does not even support dynamic invocations.

Reflection frameworks for Java (e.g. Kava [27], Reflex [26] and Javassist [5]) complement the existing Java reflection APIs by adding further reflection support at load-time through byte-code instrumentation. This load-time approach has been popularized by Java thanks to its inherent dynamic code loading and linking facilities. A limited mecha-



nism for behavioral reflection with a load-time flavor has been officially added to Java at 1.3 through *dynamic proxies* [25]. A dynamic proxy is a program-controlled typed proxy object implementing a set of types chosen by the program. It is created at runtime by generating a corresponding proxy class directly as byte-code and loading it. Due to its late addition, and inherent limitations of such an extension approach (the proxy class implements the types it is created for, methods simply reify invocations to them) this approach however only supports interface types [9].

Similarly, neither the inherent introspection capabilities nor any of the above-mentioned frameworks provide support for generics due to their late addition to the Java language. The Erl-G infrastructure handles generics by default. Erl-G also differs from the Java-related frameworks in that it generates the reflective code and allows programmers to modify it and apply further transformations to regular Eiffel code, providing more flexibility and also safety (e.g., byte code additions and modifications end up being visible through introspection in Java). On the downside, Erl-G does not yet cover behavioral reflection.

8 CONCLUSIONS

Reflection has become a fundamental feature of programming languages, not only object-oriented ones, going as far as impacting the very design and underlying computation model of recent languages, and even advocating for novel software development paradigms [16].

Many programming languages however have been initially introduced without reflection capabilities, or have been extended after their inception with features not covered by the initial reflection mechanisms. In particular languages compiled to native code, devoid of a standardized compilation and runtime infrastructure, are very prone to such extensions remaining beyond the reach of reflection. Even Java, in spite of its “compile once, run everywhere” credo, is a popular example of such a shortage: it has been augmented with generics lately but lacks support for introspection of corresponding generic parameters.

This paper proposes a way out of such bottlenecks, by describing an approach for the generation of introspection libraries leveraging static program analysis. We have described our approach starting from a general context – a simple abstract language including generics. We have presented two variants of our generation function, depending on whether the type system of the target language has a universal root type (e.g., Java, Oberon) or not (e.g., C++, Ada’95). We discussed consequences of these options in qualitative as well as quantitative terms – showing an order of magnitude gained in both time (duration of program compilation) and space (number of meta-classes generated) when a universal root type exists. We presented workarounds for limitations such as infinite recursion, and discussed minimal support required from a target programming language.

The Eiffel language has been used in this paper as an illustration for all of our contributions, as it has yielded the initial motivation for this work: only a very recent overhaul of the language has provided its type system with a universal root type, and several in-



dependent, highly evolved, compilers exist for it. We have illustrated the benefits of our approach in the Eiffel language for instance through AutoTest, an automatic program testing tool. We are currently in the process of validating our approach with other languages, and extending Erl-G with behavioral reflection.

REFERENCES

- [1] ARNOLD, K., AND GOSLING, J. *The Java programming language*. The Java Series. Addison-Wesley, 1996.
- [2] BEZAULT, E. *Gobo Eiffel tools library*.
<http://www.gobosoft.com/>.
- [3] BOBROW, D. G., DEMICHEL, L. G., GABRIEL, R. P., KEENE, S. E., KICZALES, G., AND MOON, D. A. Common Lisp object system specification. *ACM SIGPLAN Notices* 23, SI (1988), 1–142.
- [4] CHIBA, S. A metaobject protocol for C++. *ACM SIGPLAN Notices* 30, 10 (1995), 285–299.
- [5] CHIBA, S. Load-time structural reflection in Java. In *Proceedings of ECOOP 2000* (2000), vol. 1850 of LNCS, pp. 313–336.
- [6] CIUPA, I., AND LEITNER, A. Automatic testing based on design by contract. In *Proceedings of the Net.ObjectDays 2005* (2005), pp. 545–557.
- [7] CIUPA, I., AND LEITNER, A. *AutoTest: a contract-based testing tool*.
http://se.ethz.ch/people/leitner/auto_test/, 2005.
- [8] D. RIVIÈRES, J., AND SMITH, B. The implementation of procedurally reflective languages. In *LISP and Functional Programming* (1984), pp. 331–347.
- [9] EUGSTER, P. Uniform proxies for Java. In *Proceedings of OOPSLA 2006* (2006), pp. 139–152.
- [10] GOLDBERG, A., AND ROBSON, A. *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983.
- [11] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *The Java language specification, Third Edition*. Addison-Wesley, 2005.
- [12] GOWING, B., AND CAHILL, V. Meta-object protocols for C++: the Iguana approach. In *Proceedings of Reflection '96* (1996), pp. 137–152.
- [13] HEJLSBERG, A., GOLDE, P., AND WILTAMUTH, S. *C# language specification*. Addison-Wesley, 2003.

- [14] ISHIKAWA, Y., HORI, A., SATO, M., MATSUDA, M., NOLTE, J., TEZUKA, H., KONAKA, H., MAEDA, M., AND KUBOTA, K. Design and implementation of metalevel architecture in C++ – MPC++ approach –. In *Proceedings of Reflection'96* (1996), pp. 141–154.
- [15] KICZALES, G., DES RIVIÈRES, J., AND BOBROW, D. *The Art of the metaobject protocol*. MIT Press, 1991.
- [16] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. An overview of AspectJ. In *Proceedings of ECOOP 2001* (2001), pp. 327–353.
- [17] LEITNER, A. *ERL-G: the Eiffel reflection library generator*. http://se.inf.ethz.ch/people/leitner/erl_g/, 2005.
- [18] LEITNER, A., EUGSTER, P., ORIOL, M., CIUPA, I. Reflecting on Eiffel. Technical report under submission
- [19] MEYER, B. Applying design by contract. *IEEE Computer* 25, 10 (1992), 40–51.
- [20] MEYER, B. *Eiffel: the language*. Object-Oriented Series. Prentice-Hall, 1992.
- [21] MEYER, B. *Object-oriented software construction*, 2nd ed. Prentice-Hall, 1998.
- [22] PRIES, J. Objective-c. *J. Object Oriented Program.* 1, 5 (1989), 77–80.
- [23] RIVARD, F. Smalltalk: a reflective language. In *Proceedings of Reflection'96* (1996), pp. 21–38.
- [24] STROUSTRUP, B. *The C++ programming language*. Addison-Wesley, 1986.
- [25] SUN. *Dynamic proxy classes*, 1999.
- [26] TANTER, É., BOURAQADI, N., AND NOYÉ, J. Reflex — towards an open reflective extension of Java. In *Proceedings of Reflection 2001* (2001), pp. 25–43.
- [27] WELCH, I., AND STROUD, R. Kava-using byte code rewriting to add behavioural reflection to Java. In *Proceedings of COOTS'01* (2001), pp. 119–130.

ABOUT THE AUTHORS



Andreas Leitner is a PhD student and research assistant at the Chair of Software Engineering at the ETH Zurich, Switzerland. His research focus is on contract based testing and software engineering in general. He received his masters degree from Graz University of Technology, Austria, in January 2005. He can be reached at andreas.leitner@inf.ethz.ch.



Patrick Eugster Patrick is an assistant professor at Purdue University. He is interested in distributed programming, touching upon programming abstractions and languages, middleware, and distributed systems. Patrick holds both an M.S. and Ph.D. degree from the Swiss Federal Institute of Technology in Lausanne, Switzerland (EPFL), and is a recipient of the NSF CAREER award. He can be reached at p@cs.purdue.edu.



Manuel Oriol Manuel is a postdoctoral researcher at the Chair of Software Engineering at ETH Zurich. His main areas of interest are coordination, middleware, components infrastructures, dynamic software updating and software testing. He can be reached at moriol@inf.ethz.ch.



Ilinca Ciupa Ilinca Ciupa is a PhD student at the Chair of Software Engineering working on developing and evaluating techniques for automated software testing based on the principles of Design by Contract. She became a PhD student at the ETH in July 2005, after being an academic guest in the group since October 2004 and graduating as Dipl. Eng. from the Technical University of Cluj-Napoca, Romania, in September 2004. She can be reached at ilince.ciupa@inf.ethz.ch.