INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Declarative Specialization of Object-Oriented Programs

Eugen N. Volanschi, Charles Consel,

Gilles Muller, Crispin Cowan

N° 3118

Février 1997

THÈME 2

Rapport de recherche

ISSN 0249-6399

Declarative Specialization of Object-Oriented Programs

Eugen N. Volanschi, Charles Consel,
Gilles Muller, Crispin Cowan*

Thème 2 — Génie logiciel
et calcul symbolique
Projet LANDE

Rapport de recherche n°3118 — Février 1997 — 24 pages

Abstract: Designing and implementing generic software components is encouraged by languages such as object-oriented ones and commonly advocated in most application areas. Generic software components have many advantages among which the most important is reusability. However, it comes at a price: genericity often incurs a loss of efficiency.

This paper presents an approach aimed at reconciling genericity and efficiency. To do so, we introduce declarations to the Java language to enable a programmer to specify how generic programs should be specialized for a particular usage pattern. Our approach has been implemented as a compiler from our extended language into standard Java.

Key-words: generic components, program specialization, aspect-oriented programming, software engineering practices

(Résumé : tsvp)

This research is supported in part by France Telecom/CNET, contract 951W009.

* Oregon Graduate Institute of Science and Technology, *e-mail:* crispin@cse.ogi.edu

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 02 99 84 71 00 – Télécopie : (33) 02 99 84 71 71

Une approche déclarative à la spécialisation de programmes objet

Résumé : La conception et l'implémentation de composants logiciels génériques est recommandée dans beaucoup de domaines d'applications. En particulier, la conception de tels composants est facilitée par les langages objet. Les composants logiciels génériques offrent de nombreux avantages, dont le plus important est la réutilisation. Cependant, le prix à payer pour ces avantages est que la généralité induit souvent une perte de performance.

Ce rapport présente une approche permettant de réconcilier la généralité et l'efficacité. Nous proposons d'ajouter des déclarations au langage Java afin de permettre au programmeur de spécifier comment adapter un programme générique à un contexte d'utilisation particulier. Notre approche a été implémentée sous la forme d'un compilateur prenant en entrée le langage Java étendu de nos constructions, et produisant en sortie du source Java standard.

Mots-clé : composants génériques, spécialisation de programmes, programmation par aspects, génie logiciel

1 Introduction

The object-oriented paradigm has well-recognized advantages for application design, and more specifically for program structure. It makes it possible to decompose an application in terms of well-defined, generic components, closely corresponding to the structure of the modeled problem. This structuring leads to a number of important software engineering improvements regarding maintainability and re-usability of code. However, these advantages often translate to a loss in performance.

The conflict between software generality and performance has long been recognized in areas such as operating systems [9] and graphics [27]. This conflict is being increasingly addressed, with success, using forms of *program specialization*. This approach consists of adapting a generic program component to a given usage context. Program specialization can lead to considerable performance gains, by eliminating from the specialized code all the aspects which do not directly concern that precise context.

Often, specialization has been performed manually by adapting critical program components to the most common usage patterns [34, 33, 6]. This manual approach solves the efficiency problem, but has a limited applicability, because of the complexity of such a task. Recently, some tools have been developed to automatically specialize programs [1, 3, 11, 4, 25, 26, 10]. Applications of such tools are emerging in a number of fields, including scientific code [4, 5, 12], systems software [19, 29], computer graphics [21, 2], with already very promising results.

In essence, these automatic tools are transformation engines: they implement a set of program transformations which may be directly accessible to the programmer [18] or generated by some program analyses based on the usage context of the program [3, 11, 26, 1]. Programs can now be efficiently specialized at either compile time [1, 11] or run time [26, 3, 11].

The variety of options now available to the programmer drastically broadens the applicability of program specialization. However, it also makes it difficult to use for a non-expert.

Some steps towards helping the programmer control specific aspects of specialization have been achieved. In some systems, annotations are introduced in the original program to delimit code fragments to be specialized, or to indicate how to manage specialized code at run time [3]. In other systems, some programming constructs in the language are overloaded to specify potential stages at which specialization can occur [26]. Although these existing strategies make specialization more usable for non-expert programmers they address very specific aspects of program specialization; many issues remain unexplored.

This paper

We present a complete declarative approach for program specialization in the context of the object-oriented paradigm. Our approach addresses some important open issues:

- *Fully integrated declarative support.* Declaring the specialization behavior of a program should not cause the disturbance of the source of this program. We advocate that declaring specialization should be seen as adding information just as one would declare different aspects of a software component in an Aspect-Oriented language [24]. In fact, in our approach, a separate specialization declaration can be associated with any class.

- *A uniform approach.* As previously noted, progress in program specialization has considerably increased the number of parameters of this process (manual, automatic, incremental, run-time, compile-time, ...). Our declarative support offers a uniform way of exploiting these various strategies.

- *Flexible execution support.* Introducing specialization in an existing application often requires mechanisms to be integrated in the run-time environment. These mechanisms include: triggering the specialization, detecting when a specialized component can no longer be used because the usage context has changed, deciding which specialized versions should be kept and how long. In our approach, some of these aspects are automatically inferred from the declarations; others are explicitly specified in the declarations.

Overview of our Approach

We have developed language extensions for object-oriented languages aimed at expressing program specialization in a separate and declarative way. In our approach, the user declares what program components should be specialized and for which usage contexts. For an object-oriented language, this amounts to specifying what methods should be specialized and what variables should be used for specialization. Given these declarations, a special-purpose compiler determines how the specialized methods will be generated and managed. The result of the compilation is an extended version of the source program, able to trigger specialization when needed, and to replace the specialized components in a transparent way.

Our unit of declaration is a *specialization class*. It enriches information regarding an existing class. The relationship between regular classes and specialization classes is defined by a form of inheritance, based on *predicate classes* as developed by Chambers [8]. An important consequence of this technique is the ability to perform incremental specialization [14] based on class inheritance. That is, the specialization of a class is not fixed, it evolves as specialization values become available.

Configuring an application for a given usage context now amounts to separately declaring a set of specialization classes. From these declarations, the specialized behavior is derived.

Our declarative approach to program specialization has been implemented for Java. It has been introduced as a preprocessing phase. Our implementation produces standard Java programs which integrate a customized execution support for specialization.

Example

Let us briefly illustrate our approach with a sketch of a specialization class in the context of a file system. Assume this file system has a class `File` which contains, among other variables, an open mode and a reference counter, two methods for reading and writing a file,

```
spec ExclAccessFile specializes class File
{
    count == 1; // no concurrent readers/writers
    mode; // open mode is known (any mode is useful)

    read(); // produce specialized versions:
    write(); // no open mode check, no file locking

}
```

Figure 1: Declaring specialization for a file

and a method for duplicating the file descriptor (`dup()`). When a file is opened, a new `File` instance is created, and the `mode` variable is set to define the permissible file operations for the whole session (read and/or write). The reference counter is set to one until the file is shared by duplicating the file descriptor. As explained by Pu *et al.* [33], the execution of read and write operations can be significantly sped up when the file is not shared and the mode is fixed (either read, write or read/write). This allows the elimination of some operations which are done at every read or write: file-level locking is not needed because of the exclusive access, and some tests depending on the open mode can be performed only once, when the file is opened. These specializations can be modeled by the specialization class in figure 1. Based on these declarations, our compiler can infer that specialization must be done at run time, and modifies the file class implementation to trigger specialization whenever the reference counter becomes 1.

Contributions

The contributions of this paper can be summarized as follows:

- We introduce a high-level language for declaring program specialization aspects of programs. This language is fully integrated in the object-oriented paradigm.
- This integration allows us to introduce incremental specialization based on the class hierarchy. As a result, classes exploit specialization values as they become available.
- Processing these declarations both produces directives to perform various forms of specialization (compile time, run time, ...) and generates the run-time support to manage the specializations. As a result, our declarative approach brings together a variety program specialization techniques, usually used in isolation.
- Our approach is implemented for the Java language. Processing of specialization declarations produces standard Java programs.

The rest of this paper is organized as follows. In section 2 we examine the aspects that should be covered by a declarative approach to specialization. In section 3, the syntax and the semantics of specialization classes are presented, a complete example of specialization classes is given, and our compilation scheme to Java is outlined. Section 4 describes specific aspects of specialization classes involving run-time specialization. Section 5 discusses related work, and Section 6 presents some perspectives and conclusions.

2 Issues in Declaring Specialization

A declarative approach to defining the specialization behavior of a program component must address all the aspects of specialization. More specifically, it should make it possible to express *what* should be specialized and *how* it should be specialized. Before presenting the details of our approach, let us first examine these aspects.

2.1 What to Specialize

Declaring specialization involves identifying the program components that should be specialized, and describing the usage contexts for which the components should be adapted. Components associated with multiple usage contexts may require incremental specialization to best exploit all available specialization values.

Program components. Traditionally, the program components considered for specialization are: the whole program, a set of procedures, a procedure, or a block of code (within a procedure). The choice of a particular entity depends on the language structure, and more importantly, on the power of a given specializer — for example, whether the specialization process is intra/inter-block or intra/inter-procedural.

Specialization context. The usage context for which a program component is specialized is a set of predicates over some parts of the program state, which hold over some period of time. Program specializers usually consider predicates consisting of equalities, that is, specialization is performed given some variables having some specific values. In general, a predicate is said to be compile-time if it can be evaluated at this early stage. Otherwise it is said to be run-time.

Besides its stage, a predicate must be classified with respect to its lifetime, *i.e.*, whether or not the predicate can change its value during run time. If its value can never change, the predicate is said to be stable. If it can be invalidated because of some state changes, it is said to be unstable. Both stable and unstable predicates correspond to actual situations found in real experiments [34, 33, 36].

Incremental specialization. A set of predicates may not be satisfied all at once. Usually, all predicates do not become true at once. Therefore, it is useful to be able to describe a sequence of specialization contexts for a given program component so that it can be specialized further as more values become available.

2.2 How to Specialize

The operational aspects of specialization involve applying specialization and integrating the specialized components into the program.

Triggering specialization. An important issue in using specialization is when to produce the specialized code. For contexts depending only on compile-time predicates, specialization can be applied either at compile time or at run time. For the other contexts, specialization must be applied at run time. In this latter case, some run-time support must detect when all the predicates in the context become valid, trigger the specialization, and replace the component (either immediately or when the component is used).

Preserving the validity of specializations. For contexts depending on unstable predicates, the run-time support must preserve the validity of the specializations. This implies detecting when the predicates are invalidated, and replacing the specialized component by one compatible with the current state.

Caching specializations. Run-time specialization requires a run-time cache to associate specialized components to their contexts, so that no duplicate specializations are produced. Due to limited space resources, one may only want to keep a few frequently used specialized components. As a result, caching techniques need to be used to manage run-time specialized components.

3 Specialization Classes

Now that we have detailed the issues involved in specializing programs, let us present our declarative approach to addressing these issues.

Specialization classes separately define the specialization behavior of existing classes. A specialization class

- defines a context for specialization,
- indicates which program components should be specialized for this context, and
- possibly selects some options for the specialization support.

We choose the granularity of a specializable program component to be the set of methods associated with a (regular) class. Each specialization class is attached to a class in the target program. Multiple specialization classes can be attached to a single class, capturing different opportunities for specialization. If these opportunities define a sequence of incremental specialization stages, the specialization classes can be extended step by step, instead of being all defined from scratch.

sc_decl = [**runtime**] **spec** *sc_name parent_decl* [*cache_decl*] **{** (*pred_decl* **;**)$^+$ (*method_decl* **;**)* **}**
parent_decl = **specializes** [**class**] *class_name* |
  extends [**spec**] *sc_name*
pred_decl = *variable_name* == *value* |
  variable_name |
  sc_name variable_name
method_decl = *method_prototype* |
  method_definition
cache_decl = **cached** *cache_strategy* [[*integer*]]
cache_strategy = **LRU** |
  Amortization |
  Priority |
  BestFit |
  ...
sc_name = *identifier*

Figure 2: The syntax of Specialization Classes

The syntax of specialization classes is given in figure 2. The syntax is given as a Java extension: the undefined non-terminals (*identifier, integer, class_name, variable_name, method_name, method_prototype* and *method_definition*) are those defined by Java.

3.1 Semantics of Specialization Classes

In this section, we describe informally the semantics of specialization classes. The compilation scheme from a Java program with specialization classes to a standard Java program is discussed in section 3.3.

In Java, there are already two forms of inheritance. First, a class (or an interface) can *extend* another class (or interface, respectively). Second, a class may also *implement* some interfaces. We tried to keep the same spirit, by attaching a specialization class to a (regular) class via a new form of inheritance, *specialize*.

All the specialization classes attached to a regular class C form a hierarchy of possible specialization states. Class C is called the *root class*. Each specialization class either specializes a regular class, or extends another specialization class.

All the fields (variables and methods) occurring in a specialization class must exist in the corresponding root class. In other words, a specialization class cannot add new fields to a class. This is because specialization classes are not meant to add functionality to a class, but to adapt it to a particular context. To define this specialization context, a specialization class S adds some constraints over existing variables of some class C, by defining a list of predicates on the object's state. An object of class C is considered to be "in specialization state S" whenever *all* the predicates of specialization class S are valid. When this is the case, the specialized methods defined by S override the generic versions defined in class C.

The predicates can either refer to the local state, if they involve an instance variable of non-object type, or to non-local state, if they include an instance variable of object type. For the moment, we do not consider array instance variables, nor class variables[1].

A predicate on a variable of non-object type can be either compile-time or run-time. The syntax for compile-time predicates is "*variable_name*`==`*value*`;`", where the value is a compile-time constant. The syntax for run-time predicates is simply "*variable_name*`;`" without giving any value for the variable. This predicate is true for any value and is valid as long as the value does not change.

A predicate on a variable of some object type `class` *C1* constrains that sub-object to be in a specific specialization state *S1* (where *S1* is a specialization class for `class` *C1*). The predicate is written "*S1 variable_name*`;`", similar to a Java type declaration. In fact, it can be considered that the variable was redefined to be of a more precise type.

We made a deliberate design choice to restrict predicates to these simple forms, which are directly usable by a program specializer. This restriction allows the specialized method to be automatically derived from the generic definition by the program specializer. However, the syntax allows the user to specify a complete method definition for the case when the method is specialized manually.

Note that there is no distinction between stable and unstable predicates, at the syntactic level. Indeed, in an object-oriented language such as Java, it is simple to determine statically which predicates can never be invalidated — it suffices to check that the corresponding variables are only assigned in the constructors.

The syntactic constructs described so far allow the user to declare what specialization to do. There are some other syntactic constructs through which the user can influence how the specialization is achieved (keywords `runtime` and `cached`). They relate to run-time specialization, and are discussed in section 4.

3.2 Specializing a Filesystem

In a previous study [33], Pu *et al.* motivated the need for incremental specialization in the context of adaptive operating systems. This experiment focused on the HP-UX file system. A number of stable and unstable predicates were identified[2], under which specialization was performed by hand with very good results. The validity of specialized versions was managed by hand-written pieces of code called *guards*, manually inserted in the modified file system source, which both installed specialized versions and restored unspecialized ones.

The example presented below is a Java program, directly inspired by the HP-UX experiment. We define the specialization classes for the main objects, and we show how our approach makes the management of specialized versions automatic.

The key concepts of a Unix-like file system are the *file* and the *i-node* (for the sake of simplicity, we do not take v-nodes into account). Two data structures implement these

[1] The target Java program can contain any array or class variables; the only restriction is that they cannot be involved in a predicate.

[2] In that study, stable predicates are called invariants, and unstable predicates are called quasi-invariants.

entities: the file descriptor and the i-node descriptor. In fact, the file concept in Unix is an abstraction for character streams. It covers much more than regular disk files, as it includes: sockets, devices, pipes, etc. The actual type of the file is stored in an i-node descriptor — there is one for each device, disk file, or pipe — that is referenced in the file descriptor.

An i-node descriptor (see figure 3) contains, among other elements, type information and access permissions. In fact, the type information (partially in the `pipe` flag and partially in the `mode` bit-mask) never changes once the i-node is created. The access permissions (also part of the `mode` bit-mask) may be changed via the `chmod()` method. The most important methods of an i-node are the read and write methods (for conciseness, `write()` is omitted). The read method is generic in that it covers functionality belonging to all types of i-nodes.

Figure 4 shows a specialization class which specializes the generic i-node definition for the case where the i-node refers to a read-only, disk file. It declares that whenever an i-node has the specified values for the variables `pipe` and `mode`, a read operation should invoke a read method specialized with respect to these values. Whenever the state changes (*i.e.*, the i-node is no longer read-only), the generic version should be used again.

One could advocate for a different design of the filesystem, which consists of creating a static i-node sub-hierarchy, with a different class for pipes or devices; this would make it possible to separate the read functionality into distinct, overloaded methods. This design is seldom adopted, because the different read functionalities are quite complex and closely intertwined; such a separation would create a lot of hand-written versions of the same methods, and increase the risk of errors. Program specialization is an appropriate solution here, because it allows to derive the specialized functionality for each i-node type, directly from the generic version. Program specialization can furthermore eliminate other checks which cannot be encoded in a static hierarchy, like testing the access permissions at each read operation.

3.3 Compiling Specialization Classes

The compilation scheme takes as input a Java class definition and a set of specialization classes. It creates a new Java class definition that incorporates all the specialization behavior. We detail the compilation process for the i-node example.

As a naming convention, notice that new field names (variables or methods) introduced by the compiler are prefixed with `'sc'`.

The compiler splits the functionality of the original i-node into several objects (see figure 5). An *enclosing* object (shown in figure 6) reproduces the basic functionality of the original i-node, but delegates the evolving part of its behavior to an *implementation* object. Actually, two implementations are created: a generic one (shown in figure 7) and a specialized one (shown in figure 8), corresponding to the specialization class `ReadFileInode` (see figure 4).

Such an object, which changes its implementation dynamically is abstracted by an *interface*, named `Mutable` in figure 9. Therefore, the compiled i-node is enriched to support this level of abstraction. It includes a new variable, named `scImpl`, that refers to the current implementation object. The "specializable" method `read()` in the enclosing object is replaced

```
public class Inode extends Object {
  short mode; // rwx info + device/regular file
  boolean pipe; // true if i-node is pipe
  // ...

  // constructor:
  public Inode(short mod, boolean pip) {
    mode = mod;
    pipe = pip;
  }

  // change access mode (rwx bits):
  public void chmod(short mod) {
    // ... various checks
    mode = (mode & ~RWX_BIT) | mod;
  }

  // read one byte:
  public int read() {
     // Generic version: file or pipe or device
     // ... check if readable
  }
};
```

Figure 3: The original i-node definition

```
spec ReadFileInode specializes class Inode
{
  mode == I_REGULAR | R_BIT;
  pipe == false;

  read(); /* produce a simpler version:
               disk file, read-only */
}
```

Figure 4: A specialization class for an i-node

with a forwarding method, which simply invokes the current implementation object. Note that the implementation object must execute all the methods as if they were executed by the original i-node object. To do so, the implementation object maintains a pointer (called `scEncl`) to the enclosing object. Any self reference that occurred in the original object methods (via `this`) is replaced by a reference to the enclosing object (via `scEncl`).

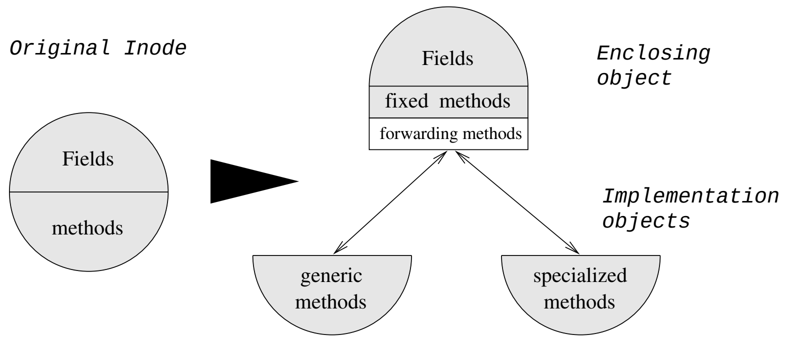


Figure 5: The compiled i-node

```
public class Inode extends Object
implements Mutable {
  // copy original fields:
  short mode; // rwx info + device/regular file
  boolean pipe; // true if i-node is pipe

  // add some new fields:
  InodeImpl scImpl; // current implementation
  List scClients; // list of Mutable clients

  // rewrite constructors:
  public Inode(short mod, boolean pip) {
    // copy original body:
    mode = mod;
    pipe = pip;

    // add initialization code:
    scClients = null;
    scNotify();
  }

  // copy original methods and rewrite
  // assignments to Mutable fields inside:
  void chmod(short mod) {
    // ... various checks
    scImpl.scSet_mode((mode & ~RWX_BIT) |
                      mod);
  }

  // add Mutable behavior:
  public void scNotify() {
    // Determine the new specialization class
    // & switch to it
    if (mode == (I_REGULAR | R_BIT) &&
        !pipe)
      scSwitchToImpl("ReadFileInode");
    else
      scSwitchToImpl("Inode");
    // propagate the notification to all clients:
    scClients.scNotify();
  }
  protected void scSwitchToImpl(String spec)
    {...}
  public void scAttach(Mutable m)
    {...} // add client
  public void scDetach(Mutable m)
    {...} // delete client
  public boolean scIsA(String spec)
    {...} // inspect state

  // forward specializable methods
  public int read() { return scImpl.read(); }
};
```

Figure 6: The enclosing i-node class

```
import sclib.*;

class InodeImpl extends Impl {
  // fixed part for a generic implementation:
  protected Inode scEncl; // the enclosing object
  public InodeImpl(Inode i) { // constructor
    scEncl = i;
  }

  // set methods:
  short scSet_mode(short new_mode) { // guarded
    scEncl.mode = new_mode; // the assignment
    if(new_mode ≠ I_REGULAR | R_BIT)
      scEncl.scNotify(); // inform the Inode
    return new_mode;
  }

  // specializable methods:
  int read() {
     // Generic version: file or pipe or device
     // ... check if readable
     // with 'scEncl' substituted for 'this'
  }
};
```

Figure 7: The generic i-node implementation

```
import sclib.*;

class ReadFileInodeImpl extends InodeImpl {
  // redefine (some) set methods:
  // (none here)

  // redefine (some) specializable methods:
  int read() {
    // ... simpler version: disk file, read-only
  }
};
```

Figure 8: The specialized i-node implementation

```
package sclib;

public interface Mutable {
   public void scNotify();
   private void scSwitchToImpl(String sc);
   public void scAttach(Mutable m);
   public void scDetach(Mutable m);
   public boolean scIsA(String sc);
};
```

Figure 9: The Mutable interface

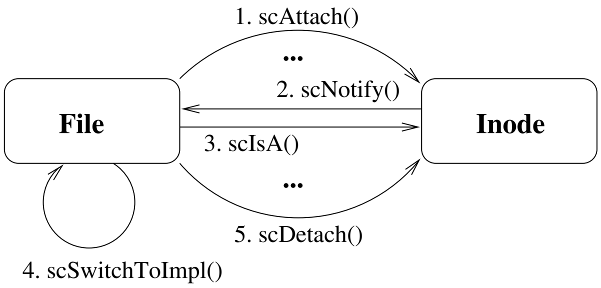


Figure 10: The protocol between `File` and `Inode`

In order to detect state changes, which may affect the current implementation, assignments to variable `mode` are *guarded*: every assignment to this variable is replaced by a call to a new method, named `scSet_mode()`. Besides doing the assignment, this method checks whether the new value invalidates specialized methods. If an assignment has invalidated the current specialization state, the `scNotify()` method is invoked to determine a new specialization class. More precisely, it proceeds by taking every specialization class related to the i-node, and checks whether all its predicates are satisfied. When a specialization class is found (possibly the generic class), the current implementation is updated by invoking the private method `scSwitchToImpl()`.

In fact, moving to a new specialization class may change the set of guarded variables since it may rely on predicates involving different variables. Note that assignments to the `pipe` variable are not guarded; a trivial static analysis determined that this variable is never assigned outside the constructors.

When changing the implementation, `scSwitchToImpl()` method produces a new instance of either `InodeImpl` or `ReadFileInodeImpl`. In the i-node example, all the specialized methods are produced at compile time, but in case of run-time specialization, creating a new implementation invokes the run-time specializer on the fly.

Assignments in the constructor(s) are *not* guarded, since the state is not yet completely initialized. The constructor(s) are only extended to call the `scNotify()` method right before returning. Indeed, at this stage the state is complete and the implementation corresponding to this initial state may be selected.

3.4 Extending the Filesystem Example

We now present a complete example, where both the file object and the i-node object are specialized. The example shows how the two specializations are *composed*, by defining a predicate on non-local state, and by preserving its validity at run time. It also covers the use of run-time predicates and an example of incremental specialization.

The original definition of the file descriptor is given in figure 11. A file descriptor is created each time a file is opened. The descriptor contains a reference to the corresponding i-node, an opening mode (which cannot be more permissive than the access mode of the i-node), the current position in the file, and a reference counter (indicating how many processes are using the file). Variables `ino` and `mode` never change after the file is opened. Variable `count` can only be changed if the file is duplicated (method `dup()`). In practice, this operation is executed quite rarely. The file position is incremented at every read.

The specialization for the i-node is the same as previously. For the file, two specialization stages are defined, as described in figure 12. The first stage (specialization class `ReadFile`) defines an open disk file: the predicate on the mode is a run-time predicate, since it holds for any value. In fact, any open mode triggers useful specialization. Variable `ino` is redefined to be of specialization class `ReadFileInode`. Thus, it requires the corresponding i-node to be in specialization state `ReadFileInode`. As a result, specialization happens inter-procedurally, that is, not only does the read method of the file object get specialized, but also the read method of the embedded i-node object. In our example, the specialized version eliminates the access test and in-lines the call to the (specialized) i-node's read method.

The second specialization stage for a file adds a new constraint: the disk file should be accessed exclusively by one process. In this case, the read method is even simpler, because the concurrency constraints on the file level can be ignored. Therefore, the `synchronized` branch is removed in the specialized version.

To keep the specialization states coherent between two inter-dependent objects (such as the file and its i-node), there is a simple protocol between the two objects via the `Mutable` interface (see figure 9). Each mutable object, such as the i-node, maintains a list of references to other mutable objects which depend on its state. When a file is connected to an i-node, by an assignment to its `ino` variable, the file declares itself as a client of the i-node object, by invoking the `ino.scAttach()` method (see figure 10). The `ino.scAttach()` method records the file in a list of clients (variable `scClient`). Consequently, if some predicates in i-node do not hold anymore, the file object is notified (`scNotify()`) that its i-node cannot be used for specialization anymore. Method `scNotify()` inspects the current state of the file, including the state of the i-node — by calling `ino.scIsA("ReadFileInode")`. The boolean method `scIsA("state_name")` allows a client to check if a mutable object has at least reached a given specialization state. Finally, when a file is closed, it invokes `ino.scDetach()`, which deletes that file from the `scClient` list.

Let us examine how the whole example works with a simple main program (file's `main()` method), as shown in figure 11. The program creates an i-node for a disk file, initially with read/write permissions, and opens a file object on this i-node in read-only mode. The constructors for both objects inspect their states. The i-node selects the generic implementation, because the access permissions are not read-only. The file selects the generic implementation too, because its i-node is not in state `ReadFileInode`.

Afterwards, the access permissions of the i-node are changed to read-only. Since the assignment to the `mode` variable is guarded, the predicate defined on `mode` is checked. Because it now holds, the i-node object switches to a `ReadFileInode` implementation, and notifies the file object. The file notices that the i-node is now specialized, and switches directly to `ExlcReadFile`, because the value of variable `count` is 1.

Finally, a `dup()` operation applied to the file triggers an un-specialization of the file up to the `ReadFile` specialization class. When the i-node returns to a read-write mode, both objects return to the generic implementation: first the i-node object, and then, by propagation, the file object too.

4 Run-time Specialization Issues

When a specialization class defines a run-time predicate (like the predicate on the mode in a `ReadFile`), the specialized implementations for this class cannot be generated before execution because the precise constant values are not known in advance.

In contrast, compile-time predicates can be exploited either at compile time or at run time. Usually, compile-time specialization is preferred because it does not incur the run-time code generation cost. However, run-time specialization may have other advantages. For example, if N specialization classes are defined for a given root class, specialization at compile time has to speculatively generate N implementations. Run-time specialization generates only the implementations used at a given moment. Either choosing compile-time or run-time specialization is a tradeoff between time and space; this choice is usually driven by the application. As explained below, our solution consist of using a default behavior, and provide the user with declaration support to overwrite it.

Our compiler computes for any specialization class a *specialization time* attribute, which can be either run-time or compile-time. This attribute is attached to the whole specialization class (instead of to every predicate), because predicates are not exploited individually. By default, if a specialization class contains only compile-time predicates, the compiler infers a "compile-time" attribute; any other specialization class is run-time. The user can force a compile-time class to be classified run-time by prefixing the specialization class declaration with the keyword `runtime`. There is no possible keyword for compile-time, because run-time predicates cannot be exploited at compile time.

The run-time attribute is inherited. That is, any specialization class D derived from a run-time specialization class R, is run-time as well. This is because class D inherits all the predicates from class R, which are (or were forced to be) run-time.

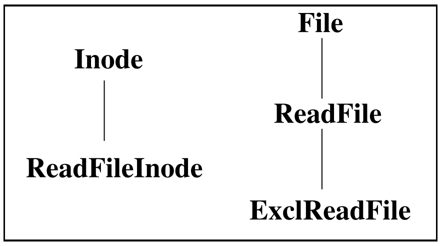
Within a single program, run-time specialization classes can coexist with compile-time ones. Furthermore, different implementations of the same root class can have different specialization times. For example, the generic implementation is always generated at compile-time, while some of its descendants can be generated at run time. In our example, the i-node implementations are compile-time, whereas the two specialized file implementations are run-time.

4.1 Caching Specialized Implementations

Caching run-time specialized implementations consists of keeping the generated implementations in a data structure of bounded size. Besides limiting the code growth, this organization has two potential benefits. The first one is to save space by avoiding to duplicate an existing

```
public class File extends Object {
  Inode ino; // reference to the inode descriptor
  short mode; // open mode (r and/or w)
  long pos; // current position in file
  int count; // No. of processes using this file descriptor

  public File(short mod, Inode inod) { // constructor
    mode = mod;
    ino = inod;
    count = 1;
    pos = 0;
  }

  File dup() {
    count++;
    return this;
  }

  public static void main(String argv[]) {
    Inode i = new Inode(Inode.I_REGULAR |
                        Inode.R_BIT | Inode.W_BIT,
                        false);
    File f = new File(R_BIT, i);

    i.chmod(Inode.R_BIT);
    f.dup();
    i.chmod(Inode.R_BIT | Inode.W_BIT);
  }

  public int read() {
    if((mode & R_BIT) == 0)
      return new FileAccessError();
    if(count > 1)
      synchronized(this) { // lock the file descriptor
        pos += 4; // Advance with the size of an integer
      }
    else pos +=4; // avoid lock when possible
    return ino.read();
  }
};
```

Figure 11: The original File definition and the main program




```
// Inode hierarchy:

spec ReadFileInode specializes class
Inode {
  mode == I_REGULAR | R_BIT;
  pipe == false;

  read(); /* produce a simpler version:
                disk file, read-only */
}


// File hierarchy:

spec ReadFile specializes class File {
  mode; /* any value is good */
  ReadFileInode ino; /* restriction on
                          inode's state */

  read(); /* produce a simpler version:
             disk file, mode is known */
}

spec ExclReadFile extends spec
ReadFile
        cached Amortization[20] {
  count == 1;

  read(); /* produce an even simpler
              version: no locking */
}
```

Figure 12: The specialization classes for the filesystem

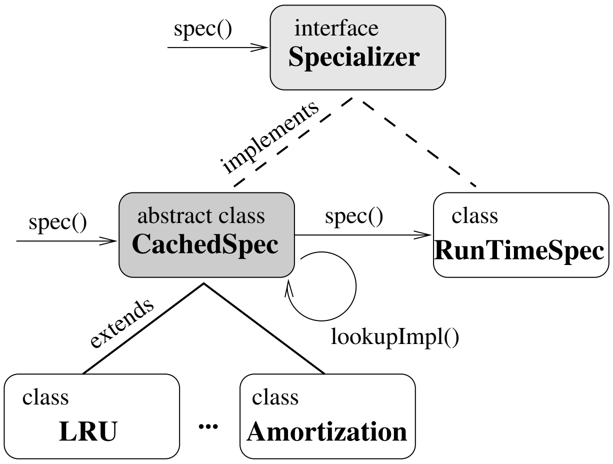


Figure 13: The cache hierarchy

implementation, whenever possible. For example, two different files which are in specialization state `ReadFile` can share the same implementation if they have the same mode.

The second benefit of caching is to save time by keeping specialized implementations that are not currently used, but are likely to be reused. For example, a directory file is likely to be opened several times. Keeping its implementation for some time after the file is closed may pay off.

Different applications may need different caching strategies. Our compiler gives a default strategy, and provides a number of alternatives. The user can choose between these alternatives, allocate new caches, and define some new strategies. To select a specific cache behavior for some specialization class S, the declaration fragment "`cached` *strategy*`[`*size*`]`" must be included in S's definition (see figure 12). This fragment specifies that a cache of `size` elements is reserved for class S. Cache behavior is also inherited, so that this cache is also used for classes derived from S (unless they redefine it).

Our implementation is extensible (see figure 13). There is a `Specializer` interface, which defines the specialization functionality (method `spec()`). The run-time specializer implements this interface. A cache is a "filter" for the run-time specializer. That is, the cache both implements the interface, and contains a reference to the run-time specializer. It satisfies some of the specialization requests by itself, if it can reuse an existing implementation from the cache; if not, it invokes the runtime specializer. In this scheme, the user can provide its own cache strategy by extending the `Cache` abstract class.

4.2 Some Caching Strategies

A number of general cache strategies (*e.g.*, LRU, multi-way associative, etc.) can be used for the specialization caches. However, caching of run-time specialized implementations raises some specific issues:

- Specialization must be *amortized.* Failing to take this constraint into account may trigger frequent specializations which will never pay off.
- Specialization classes are *ordered* by the "more specialized than" relation. The most specialized implementation is intended to give the best performance gain. There is also some compatibility between ordered implementations. That is, an implementation for a given specialization class is applicable to any derived specialization classes.
- A specialization request can be refused, when resources are unavailable, by defaulting to the generic implementation.

Taking these issues into account, some novel, domain-specific strategies can be studied. Let us outline some examples.

- *Amortization:* Considers the amortization effect as the most important aspect. Therefore, a specialized implementation is never evicted from the cache until it is no more referenced. This strategy can cause some specialization opportunities to be neglected when space is exhausted.
- *Depth-most priority:* Specialization classes are prioritized upon their "depth" (or degree) of specialization in the inheritance tree. In case of an eviction from the cache, the least specialized implementations is chosen. This strategy makes highly specialized methods more likely to be available
- *First fit:* In response to a specialization request, this strategy searches in the cache until if finds any implementation compatible with the requested one. Implementations are evicted only when no approximate fit is found. This strategy encourages re-use of implementations, at the price of approximate fitting.

4.3 Overhead of the Execution Support

Managing specialized version at run time imposes some overhead, both in terms of time and space. Our execution support tries to minimize the time overhead in frequently executed paths. Namely, as long as the state does not change, each call of a specializable method incurs only one extra method call to delegate the behavior to the current implementation. Only when there is a state change, the execution support is invoked to perform various tasks: determining which specialization to choose, calling the specializer on-the-fly (for run-time classes), propagating notifications, and replacing the implementation.

Furthermore, the overhead is not paid by all objects of a root class. In the filesystem example (see figure 12), assignments to variable `count` are not guarded in any generic file object because this part of the state does not affect the current implementation.

4.4 Current Status

We have implemented a compiler prototype for specialization classes, written in Java. We are currently working on a specializer for Java code which is not yet fully functional. The idea is to use our existing Java-bytecode to C compiler, called Harissa [28], as a front-end to our existing specializer for C programs, called Tempo [12, 23]. Specialized programs may then be translated back to Java.

In the meantime, one can use specialization classes in two ways. First, Tempo can be directly used for specializing *native* methods in Java programs, both for compile-time and run-time specialization classes. Second, the user can supply manually specialized code, by supplying a complete definition for each specializable function in a specialization class. Note that this manual strategy only works when the code can be specialized statically (*i.e.*, only for compile-time specialization classes).

5 Related Work

Predicate classes [8] are a form of dynamic inheritance, which complements the static inherintace of an object-oriented language. They offer support for defining an object with several implementations, which are dynamically selected based on arbitrary predicates on the object's state. Implementations are not restricted to specialization of an existing behavior; they can add completely new functionalities. All implementations must be defined statically, by hand. Part of the interface checking must be done at run time, unless avoided by some user annotations. The dynamic selection of implementations is based on multi-dispatching.

Specialization classes are based on predicates classes, but are tailored for a particular goal: program specialization. Our predicates have some simple forms, which allows us to automatically derive implementations, even at run-time. As the interface of specialized objects never changes, static interface checking can still be done without any user guidance. The assumption that unstable predicates seldom change allows us to optimize the implementation, and to fully integrate it into a single-dispatched language such as Java.

Dean *et al.* explore a specific form of program specialization, which is aimed at eliminating most of the virtual method calls in an object-oriented language [17]. To eliminate a virtual method call on a receiver object, a clone of the caller method is produced for each possible type of both the receiver and the arguments. Special care is taken to avoid an uncontrolled code growth by selectively performing this optimization. The selection algorithm estimates the impact of a potential specialization based on profiling information. In their approach, methods are specialized only with respect to the *type* of the objects. Our specialization is with respect to the state of the objects. An interesting direction would be to explore a selection algorithm for (general) program specialization. This algorithm would compute the most useful specialization classes, based on profiling information.

Existing program specializers offer different levels of support for expressing and guiding specialization. In CMix [1], a specializer for C programs, the transformations are guided by command line parameters to the specializer. The management of specialized components is not addressed, because the program is always specialized as a whole. In Tempo [11], (our specializer for C programs) both run-time and compile-time specialization can be done [13], based on a separate description of the specialization context. This description is flat and somewhat limited, due to the lack of sufficient structure in C programs. The run-time specialized versions must be entirely managed by the user. In Fabius [26], a specializer for ML programs, the user guides the transformations by rewriting the program to expose two execution stages, using currying. The specializable component is a function, and it is up to the programmer to generate and manage the specialized functions. In the C dynamic compiler developed by University of Washington [3], the programmer can mark replaceable components by directly annotating the program, using a few syntax extensions. The management of specialized blocks is done automatically.

Cowan *et al.* describe some execution support for managing several versions of the same procedure, in the context of adaptive operating systems [15]. Their work focuses on a replugging algorithm able to deal with the concurrency issues of an operating system.

Finally, let us note that our compiling scheme for generating the execution support generalizes some programming patterns found in object-oriented applications dealing with forms of dynamic adaptivity. As an example, in the standard Java library, several objects offer some adaptivity to security restrictions. The `java.net.Socket` class delegates part of its functionality to an implementation object called `SocketImpl`. These implementation objects are created by a `SocketImplFactory` object, which can be replaced by a user-defined one. This adaptivity is irreversible in that once a socket is created, its implementation cannot change anymore.

6 Future Work and Conclusions

Specialization classes provide a complete declarative approach to describe the specialization behavior of program components. As a result, specialization can better adapt a program to a specific usage pattern. Specialization classes allow the programmer to declare what program components should be specialized and how they should be specialized. This approach uniformly captures the numerous emerging alternatives to perform specialization (compile time, run time, incremental, automatic, ...).

Specialization classes are fully integrated in the object-oriented paradigm. It declares specialization aspects of existing classes without disturbing the source program.

Specialization classes define a specialization context which can be applied at both compile time and run time. If a specialization context changes because of some state updates, this situation is automatically detected and the corresponding specialization methods are either specialized with more values or made more generic if some specialization values become unavailable. Because specialization classes form a class hierarchy, objects can be specialized incrementally as more specialization values become available.

Specialization classes allow the programmer to specify what run-time support a class should include to manage its specializations. In particular, various cache strategies can be declared to better fit the specialization needs of the application.

We have implemented a compiler from extended Java, with specialization classes, to standard Java. We are currently completing a program specializer for Java programs. It will then be interfaced with our specialization class compiler. It will allow a programmer to specify both compile-time and run-time specialization aspects of classes.

We are studying the generalization of the form of predicates used in specialization classes. In particular, we plan on introducing disjunctions and class variables to improve the expressive power of predicates.

Regarding applications of specialization classes, we are continuing the effort initiated by Cowan *et al.* in the area of adaptive operating system components [16]. We would like to use specialization classes to specify a number of systems optimizations that have been described in the litterature (*e.g.*, [22, 7, 35, 20]). In particular, we are redesigning a part of the Chorus IPC subsystem, to exploit opportunities for run-time optimization that have not been addressed so far because of a lack of appropriate methodologies and tools.

References

[1] L. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.

[2] P. Andersen. Partial evaluation applied to ray tracing. DIKU Research Report 95/2, DIKU, University of Copenhagen, Denmark, 1995.

[3] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In PLDI96 [31], pages 149–159.

[4] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132. Technical Report 94/9, University of Melbourne, Australia, 1994.

[5] A. Berlin. Partial evaluation applied to numerical computation. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, Nice, France, 1990. ACM Press.

[6] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[7] R. Campbell, N. Islam, P. Madany, and D. Raila. Designing and implementing Choices: an object-oriented system in C++. *Communications of the ACM*, 1993.

[8] C. Chambers. Predicate classes. In *ECOOP'93 Conference Proceedings*, Kaiserstautern, Germany, July 1993. Also published as Technical report at University of Washington.

[9] W. Cheung and L. A. Exploring issues of operating systems structuring: from microkernel to extensible systems. *ACM Operating Systems Reviews*, 29(4):4–16, Oct. 1995.

[10] C. Consel. A tour of Schism. In PEPM93 [30], pages 66–77.

[11] C. Consel, L. Hornof, F. Noël, J. Noyé, and E. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, Feb. 1996.

[12] C. Consel, L. Hornof, F. Noël, J. Noyé, and E. Volanschi. A uniform approach for compile-time and run-time specialization. Rapport de recherche 2775, Inria, Rennes, France, Jan. 1996.

[13] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In POPL96 [32], pages 145–156.

[14] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In PEPM93 [30], pages 44–46. Invited paper.

[15] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *International Conference on Configurable Distributed Systems*, Annapolis, MD, May 1996.

[16] C. Cowan, A. Black, C. Krasic, C. Pu, J. Walpole, C. Consel, and E. Volanschi. Specialization classes: An object framework for specialization. In *Fifth IEEE International Workshop on Object-Orientation in Operating Systems*, Seattle, Washington, Oct. 1996.

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

[18] D. Engler, W. Hsieh, and M. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In POPL96 [32], pages 131–144.

[19] D. Engler and M. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM Symposium on Communications Architectures and Protocols*, Stanford University, CA, Aug. 1996. ACM Press.

[20] M. Gien. Evolution of the CHORUS open microkernel architecture: The STREAM project. In *Proceedings of Fifth IEEE Workshop on Future Trends in Distributed Computing Systems (FTDCS'95)*, Cheju Island, Korea, Aug. 1995. IEEE Computer Society Press. Also Technical Report CS/TR-95-107, Chorus Systemes.

[21] B. Guenter, T. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.

[22] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. Technical Report SMLI TR-93-14, Sun Microsystems Laboratories, Inc., Apr. 1993.

[23] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press. To appear.

[24] G. Kiczales. Aspect-oriented programming. http://www.parc.xerox.com/spl/projects/-aop/, 1996.

[25] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. Fortran program specialization. In U. M. G. Snelting, editor, *Workshop Semantikgestützte Analyse, Entwicklung und Generierung von Programmen*, pages 45–54. Justus-Liebig-Universität, Giessen, Germany, 1994. Report No. 9402.

[26] P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI96 [31], pages 137–148.

[27] B. Locanthi. Fast bitblt() with asm() and cpp. In *European UNIX Systems User Group Conference Proceedings*, pages 243–259, AT&T Bell Laboratories, Murray Hill, Sept. 1987. EUUG.

[28] G. Muller, B. Moura, F. Bellard, and C. Consel. JIT *vs.* offline compilers: Limits and benefits of bytecode compilation. Rapport de recherche 1063, Irisa, Rennes, France, Dec. 1996.

[29] G. Muller, E. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing a commercial RPC protocol. Rapport de recherche 1068, Irisa, Rennes, France, Dec. 1996. To appear in ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation, 1997.

[30] *Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, Denmark, June 1993. ACM Press.

[31] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation.* ACM SIGPLAN Notices, 31(5), May 1996.

[32] *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, St. Petersburg Beach, FL, USA, Jan. 1996. ACM Press.

[33] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 314–324, Copper Mountain Resort, CO, USA, Dec. 1995. ACM Operating Systems Reviews, 29(5),ACM Press.

[34] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[35] V. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *USENIX - Workshop Proceedings - Micro-kernels and Other Kernel Architectures*, pages 39–70, Seattle, WA, USA, Apr. 1992.

[36] E. Volanschi, G. Muller, and C. Consel. Safe operating system specialization: the RPC case study. In *Workshop Record of WCSSS'96 – The Inaugural Workshop on Compiler Support for Systems Software*, pages 24–28, Tucson, AZ, USA, Feb. 1996.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399