Reifying Variants in Configuration Management

JEAN-MARC JÉZÉQUEL
Irisa/CNRS

Using a solid software configuration management (SCM) is mandatory to establish and maintain the integrity of the products of a software project throughout the project's software life cycle. Even with the help of sophisticated tools, handling the various dimensions of SCM can be a daunting (and costly) task for many projects. The contribution of this article is to (1) propose a method (based on the use creational design patterns) to simplify SCM by reifying the *variants* of an object-oriented software system into language-level objects and (2) show that newly available compilation technology makes this proposal attractive with respect to performance (memory footprint and execution time) by inferring which classes are needed for a specific configuration and optimizing the generated code accordingly.

Categories and Subject Descriptors: D.2 [**Software**]: Software Engineering; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.9 [**Software Engineering**]: Management—*software configuration management*

General Terms: Design, Management, Performance

Additional Key Words and Phrases: Compilation technology, Eiffel, Mercure, object-oriented analysis and design, reifying variants, SMDS, software configuration management

1. INTRODUCTION

In software configuration management (SCM) [Estublier and Casallas 1995; Tichy 1994] *revisions* are defined as the steps a configuration item goes through over time (historical versioning), whereas *variants* are the various implementations that remain valid at a given instant in time. The reasons why a given software design may have different implementations, all valid at a given instant in time, are manifold. But the basic idea is to be able to handle *environmental* differences, from the hardware level to the marketing context (range of products) and through specificities in the

An earlier version of this article appeared in the *Proceedings of the ACM/IEEE International Conference on Software Engineering, ICSE'20, Kyoto, Japan (April 1998).*
Author's address: Campus de Beaulieu, Irisa/CNRS, Rennes, F-35042, France; email: jezequel@irisa.fr.

target operating systems, compiler differences, user preferences for graphical user interfaces (GUI) and internationalization issues. Managing all the combinations between these variability factors can soon become a nightmare: in some cases the part of the software dealing with the configuration can be as large as the software itself.

One of the most primitive "solutions" to these problems was to patch the executable program at installation time to take into account some variants. Device drivers are a more elaborate example of configurability common to almost all operating systems, but this is limited to code that run in kernel mode. Another widely used technique for making small real-time programs configurable is the static configuration table. Since they are purely declarative with no language-defined semantics, constraint verification and consistency checking can be difficult, let alone error checking. For larger systems, one of the most popular approach consists in using conditional compilation (or assembly), implemented with, for example, a preprocessor like CPP. But this kind of code can rapidly become difficult to maintain [Ray 1995]. Static and dynamic configuration informations are completely intermingled, which makes it hard to change one's mind on what should be static or dynamic. Furthermore, to add support for a new OS, one needs to review *all* the already written code looking for relevant #ifdef parts. Even with the help of sophisticated tools [Leblang 1994], handling the various dimensions of SCM can be a daunting (and costly) task for many projects.

The contribution of this article is to (1) propose a method to simplify SCM by reifying the *variants* of an object-oriented software system into language-level objects (Section 2) and (2) show that newly available compilation technology makes this proposal attractive with respect to performance (memory footprint and execution time) by inferring which classes are needed for a specific configuration and optimizing the generated code accordingly (Section 3). We also discuss the interests, limitations, and drawbacks of our approach, as well as related works (Section 4) before concluding on the perspectives open by our approach.

2. REIFYING VARIANTS

2.1 Object-Oriented Modeling of Variants

Using an object-oriented analysis-and-design approach, it is natural to model the commonalities between the variants of a configuration item in an abstract class (or interface) and express the differences in concrete subclasses (each variant implements the interface in its own way).

The choice of which variant(s) to use for a configuration item can be made either at compile time or at runtime, with the very same source code: the idea is to rely on (conceptual) dynamic binding for the design of the system and don't care *now* for static versus dynamic distinction. Dynamic loaders would then be able to load only the chosen variants, or compilers could figure out which ones to load and then compile out the dynamic dispatch.

2.2 The Mercure Case Study

As a case study for evaluating the interest of our approach, we consider the Mercure project, which is an SMDS (Switched Multi-Megabits Data Service) server whose design and implementation have been described in Jézéquel [1996; 1998]. It can abstractly be described as communication software delivering, forwarding, and relaying "messages" from/to a set of network interfaces connected onto an heterogeneous distributed system.[1]

Mercure must handle variants for five configuration items: any number (V_i) of network interface boards, V_p specialized processors, V_n levels of functionality, and support for V_l languages (internationalization). Considering that a given variant of the Mercure software might be configured with support for any number of the V_i network interfaces and V_l languages, and one of V_p kinds of processors, one of the V_n levels of network management, and one of the V_g GUIs, the total number of Mercure variants is

$$V = V_p \times V_n \times V_g \times 2^{V_i + V_l - 2}$$

which, for $V_i = 16$, $V_p = 4$, $V_n = 8$, $V_g = 5$, and $V_l = 24$, gives more than several trillions possible variants (43,980,465,111,040 to be precise).

2.3 Object-Oriented Design of Mercure

In the Mercure software, consider for example the case of the network interface boards. Whatever the actual interface (ATM, DQDB, Ethernet, etc.), we must be able to poll it for incoming messages, to read them into memory buffers, to send outgoing messages, and to set various configuration parameters. So this abstract interface, valid for all kinds of network interface boards, could be expressed as an abstract class called NETDRIVER.

The idea underlying this kind of object-oriented design is that a method (such as *read_msg* in the class NETDRIVER above) has an abstractly defined behavior (e.g., read an incoming message from the lower level network interface and store it in a buffer) and several differing concrete implementations, defined in proper subclasses (e.g., NETDRIVER1, NETDRIVER2... NETDRIVERN). This way, the method can be used in a piece of code independently of the actual type of its receiver, i.e., independently of the configuration (e.g., on which kind of interface board do we actually read a message):

```
if driver.is_msg_available then
  driver.read_msg
  outgoing_address:= routing(driver.last_msg)
  if drivers.valid_address(outgoing_address) then
    forward(drivers.item(outgoing_address),
      driver.last_msg)
```

[1]The trimmed-down source code of Mercure (where only critical-path computations and configuration-management-related issues have been kept) is available at *http://www.irisa.fr/pampa/EPEE/SCM*.

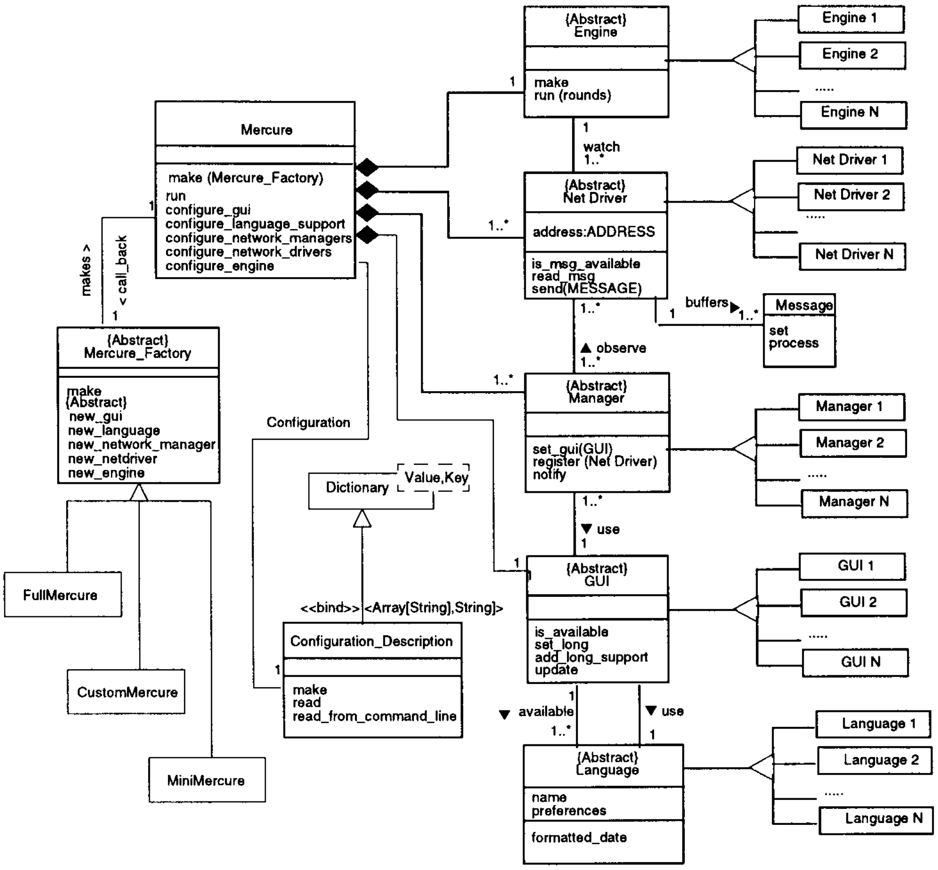

Fig. 1. Class diagram modeling the Mercure software in UML.

```
    end -- if
  end -- if
```

Using the same idea for handling the variants of the other configuration items in Mercure, we can model Mercure's design[2] as illustrated in Figure 1 through a UML (unified modeling language) class diagram. This UML diagram basically says that a Mercure system is an instance of the class of MERCURE, aggregating an ENGINE (that encapsulates the actual work that Mercure has to do on a particular processor of the target distributed system), a collection of NETDRIVERS, a collection of MANAGERS that represent the range of functionalities available, and a GUI that encapsulates the user preference variability factor. A GUI has itself a collection of supported languages, and among them, the currently selected language.

[2]On this diagram, only a rather flat inheritance hierarchy is suggested, but obviously the designer should factorize the commonalities between subclasses in an inheritance graph as deeply as required.

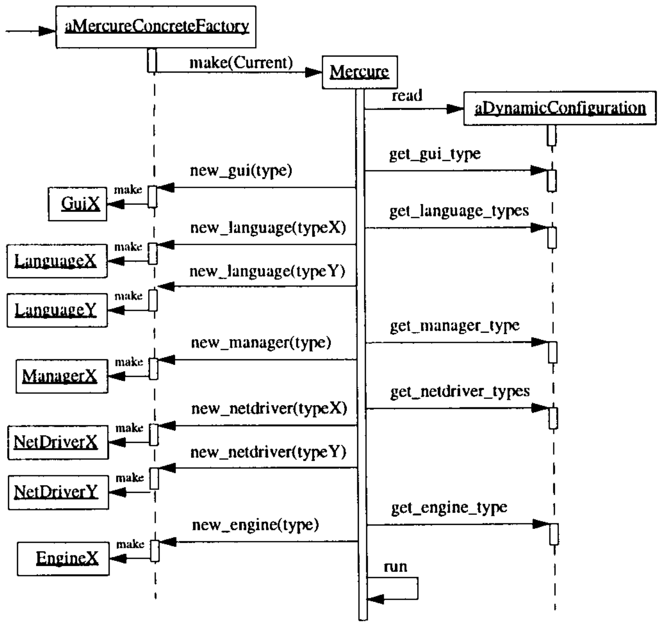

Fig. 2. Dynamic configuration of the Mercure software (UML sequence diagram).

2.4 Applying Creational Design Patterns

With this design framework, the actual configuration management can be programmed *within* the target language: it boils down to only create the class instances relevant to each configuration item of a given configuration. *Creational Patterns* as proposed in Gamma et al. [1995] can help make this process easily customizable by uncoupling the system from how its constituent objects get created, composed and represented. In our simple case, we use an *Abstract Factory*, called MERCURE_FACTORY to define an interface for creating variants of Mercure's five configuration items.

The class MERCURE_FACTORY features one *Factory Method* for each of our five variability factors. The Factory Methods are parameterized to let them create various kinds of products (i.e., variants of a type), according to the dynamic Mercure configuration selected at runtime. These Factory Methods are abstractly defined in the class MERCURE FACTORY and are given concrete implementations in its subclasses—e.g., FullMercure, CustomMercure, and MiniMercure—(see Figure 1) called *concrete factories*. A concrete factory starts by creating a MERCURE instance, which calls back the concrete factory to configure its components (see Figure 2).

Obtaining an actual variant of the Mercure software then consists in implementing the relevant concrete factory. By restricting at compile time, i.e., in the source code of a concrete factory the range of products that a Factory Method can dynamically create, we can choose to build specialized versions of the general-purpose Mercure software.

The selection of a given concrete Mercure factory as the application entry point allows the designer to specify the Mercure variant he or she wants. Since this is done at compile time, it should be possible to generate an executable code *specialized* toward the selected Mercure variant. In the next section, we show how this can be done automatically with current compiler technology.

3. COMPILATION TECHNOLOGY AND PERFORMANCE RESULTS

3.1 Principle of Type Inference and Code Specialization

Compilation techniques based on *type inference* consist in *statically* computing the set of dynamic types a *reference* may assume at a given method call site. In the most favorable case this set is a singleton, and thus the method can be statically bound, and even in-lined in the caller context. In less favorable cases, the set may contain several types. However, the compiler is still able to compute the reduced set of methods that are potentially concerned, and generate specialized code accordingly. This can be implemented as an *if-then-else* block or a switch on the possible dynamic types of the reference to select the relevant method to call. In either case, the cost of the (conceptual) dynamic dispatch can be mostly optimized out (and the cache miss implied by dynamic binding is no longer a fatality).

This idea is implemented for example in GNU SmallEiffel [Zendra et al. 1997], a free Eiffel compiler distributed under the terms of the GNU General Public License as published by the Free Software Foundation.[3] So we have implemented the Mercure software with Eiffel and used the SmallEiffel compiler to take a number of measures. Eiffel [Meyer 1992] is a pure OO language featuring multiple inheritance, static typing and dynamic binding, genericity, garbage collection, a disciplined exception mechanism, and an integrated use of assertions to help specify software correctness properties in the context of *design by contract*. However, our approach is not really dependent on Eiffel and could be applied to any class-based languages without dynamic class creation, e.g., C++ (with the compiler optimization known as the *elimination of virtual*), Ada95, or Java.

3.2 Specialized Code Generation in SmallEiffel

In Eiffel, the programmer specifies the entry point of an application by singling out a specific class (the root class, here a given concrete Mercure factory) along with one of its creation procedures.[4] When a program is run, a single instance of the root class is created (the root object), and the specified creation procedure is called. In our case, it leads to the execution trace presented in Figure 2.

To compile such a program, the GNU SmallEiffel compiler starts by computing which parts of the Eiffel source code may or may not be reached

[3]GNU SmallEiffel can be downloaded from http://www.loria.fr/projets/SmallEiffel.
[4]A creation procedure corresponds to a class constructor in the C++ terminology.

from the application root, classifying them as *living code*, i.e., code which may be executed, and *dead code*, i.e., which cannot be executed within this configuration (see Zendra et al. [1997] for more details). By analogy with the terms dead and living code, a *living type* is a class for which there is at least one instantiation instruction in the living code. Conversely, a *dead type* is a class for which no instance may be created in living code.

Code is generated in C for living types only. An object is classically represented by a structure (C `struct`) whose first field is the number that identifies the corresponding living type (corresponding to the C++ RTTI).

A Dynamic Type Set can thus be represented as a set of integers. Taking back the example of calling the method *read_msg* on a NETDRIVER object, the generated C code has the following structure:

```
void XrNETDRIVERread_msg(void *C) /* C represents the receiver */
  int id=((T0*)C)->id; /* the Dynamic Type Identifier, aka RTTI */
  switch (id) {
    /* call read_msg defined in NETDRIVER1, whose RTTI is 123 */
    case 123: rNETDRIVER1read_msg((NETDRIVER1*)C); break;
    /* call read_msg defined in NETDRIVER2, whose RTTI is 124 */
    case 124: rNETDRIVER2read_msg((NETDRIVER2*)C); break;
    ...
    case 138: rNETDRIVER16read_msg((NETDRIVER16*)C); break;
  }
}
```

Depending on the knowledge the compiler has on the possible actual dynamic types of the NETDRIVER, the number of branches in the switch can be reduced, or the switch itself could even be replaced by a sequence of *if-then-else*. If the Dynamic Type Set for the receiver is a singleton (e.g., NETDRIVER5 whose id is 127), GNU SmallEiffel further specializes the generated code by optimizing out the runtime type test completely and substituting the call to `XrNETDRIVERread_msg` by a direct call to `rNETDRIVER5read_msg`.

3.3 Performance Results for Mercure

We consider three versions of Mercure to compare the effect of the specialization of the code generation: *FullMercure*, the general-purpose version of the program (where *anyone* of the trillions of Mercure variants can be dynamically configurated at runtime), a more restricted version called *CustomMercure*, and *MiniMercure*, a minimal version of the software, with only one of each configurable part available. These three variants use exactly the same software baseline. The only difference is that a different Mercure concrete factory is selected as the root class, i.e., the class containing the entry point of the application (see Figure 1).

The left part of Figure 3 compares various compile time statistics for these three Mercure versions: the number of lines of code (LOC) for describing the configuration (i.e., the number of LOC of the relevant Mercure concrete factory, ranging from 36 to 96), the size of the full Eiffel source code (in kilolines of code), the size of the generated C code (kLOC), and finally, the type inference score. This score is the ratio of dynamic calls

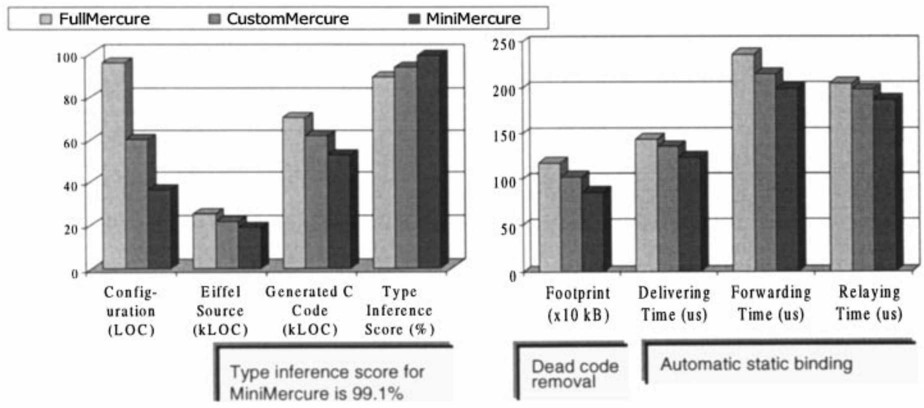

Fig. 3. Compile time and runtime statistics.

that could be replaced by direct calls at compile time. It ranges from 91% to more than 99% in MiniMercure. This means that the GNU SmallEiffel compiler has been able to early bind most of the (conceptually) dynamic binding in the MiniMercure version.

Since the dynamic configuration we choose for the Mercure and CustomMercure variants is the same as the one selected at compile time in MiniMercure, all versions have exactly the same dynamic behavior. Yet the compiler is able to use type inference to detect what is in fact configured statically in specialized versions of Mercure factories to generate a more compact and efficient code. For example, in the MiniMercure case, the call to the (pure virtual) routine *read_msg* from the class NETDRIVER has been compiled to a direct call to `rNETDRIVER5read_msg`, i.e., the version defined in the class NETDRIVER5. Actually, if we look in the generated code, we can check that it has even been in-lined in the context of the caller. More generally, the generated code has the same structure as the one that would have been obtained with a preprocessor-based method. This is illustrated in the right part of Figure 3, with the memory footprints (in tens of kilobytes) of our three versions of Mercure, as well as the mean time spent (in μs) in delivering, forwarding, and relaying SMDS messages (see Jézéquel [1998] for more details on our experimental conditions).

The substantial performance improvement (up to 18%) brought by type inference in MiniMercure can be linked to the fact that in Mercure the variants of configuration items located on the critical path are relatively small classes made of simple methods. Without optimizations, the time the system would spend into dynamic dispatch (and the related cache misses) would not be negligible. Clearly this is not always the case. For instance, we applied the ideas of this article to the configuration management of UMLAUT, a toolbox for handling UML models. In UMLAUT the configuration items are multiple parsers (CDIF, MDL, Java/C++/Eiffel, etc.) and corresponding generators, as well as the supported OS, compilers, and levels of functionality. In contrast to the Mercure case, the variants of these configuration items are rather large and complex classes. The relative time spent in optimizable dynamic dispatch is then much less important, and the overall performance improvement of specialized versions is negligible. On the other hand, the difference in footprint is much bigger: from ten to one between the general-purpose version of UMLAUT and the most specialized one.

Another way to look at the performance differences between MiniMercure and FullMercure is to consider that it represents the maximum price that the designer would have to pay for trading time and space performances for dynamic configuration capabilities. Even more interestingly, the designer can easily choose his own trade-off between these two properties: he or she has just to select the relevant concrete factory, yet keeping *the very same software baseline*.

4. DISCUSSION AND RELATED WORK

4.1 Discussion

Our approach is not the ultimate solution to all SCM problems. For one thing, it does not make variants disappear: on the contrary they are made first-class objects, so that both the engineer and the compiler can reason about it.

Since some SCM issues (variant management) are now considered at the design level, you can even show your client a UML diagram with the variants made explicit. It is then much easier and less error prone (due to static type checking) to introduce new variants of configuration items and to select and evolve them using standard OO design patterns (e.g., *Abstract Factory*).

Because the configuration is expressed in a uniform manner (no more static versus dynamic distinction at the design level), the designer can easily change his or her mind (should language selection be static or dynamic?) without changing the design, and let the compiler generate the best code out of it. This is particularily helpful for software with rapidly changing requirements.

For the compiler to reason about variants with type inference, it must have access to the full code. It is clear that in our approach we cannot deal efficiently with libraries of classes compiled in .o or .a forms. However, .o and .a Unix formats are anyway not very usable in an OO context because they lack type information. They were used in the past to solve a number of problems, that are now dealt with at another level:

—*Enforcing modularity for procedural programs:* this is now superseded by OO concepts.

—*Speed of compilation:* while this still holds for small programs, it is well known that large C++ compilations actually spend most of their time in link editing. So having .o or .a files no longer reduces much the overall edit/compile/link/test time.

—*Source protection:* having access to the full code does not mean full *source* code, because the source can be precompiled in a "distributable" format, e.g., Java *.class* formats or Eiffel "precompiled" formats from some vendors. Alternatively, sophisticated encryption technology could be used to protect the source code.

Our approach does not remove the need for classical configuration management tools. We still have to deal with *revisions* (new features, bug corrections, etc.) and possibly concurrent development activities. However, concurrent development activities are minimized by the fact that a variant part is typically small and located in its own file: someone responsible for a product variant would not have to interfere with other people modifications, and conversely. Thus in our experience, a simple tool such as RCS or CVS (equipped with automatic symbolic naming of versions) should be enough for many sites.

Doing all the configuration in the target language eliminates the need to learn and use yet another complex language used just for the configuration management. On the other hand, actually programming the concrete factories to specify the configuration is quite tedious, albeit straightforward: some kind of simple automatization scheme would be useful.

4.2 Related Work

This work can be seen as an application of ideas circulating in the "Partial Evaluation" community for years. Actually, it can be seen as taking benefit of the fact that the type of configurable parts have bounded static variations (i.e., the sets of possible types are known at compile time). Because this partial evaluation only deals with the computation of dynamic type sets, it is also clearly related with the domain of type inference [Agesen 1996].

Related work from the SCM point of view has already been extensively discussed in this article. With respect to approaches trying to leverage the object-oriented or object-based technologies, our idea of designing the application in such a way that the SCM is simplified is not new [Bendix 1992; Gallagher and Berman 1993]. But previous works needed a dedicated tool to handle the actual SCM. Since in our approach the SCM is done *within* the OO programming language, there is no need for such an ad hoc tool: the compiler itself handles all the work.

5. CONCLUSION

Our contribution in this article was to propose a method to simplify software configuration management by reifying the *variants* of configuration items in an object-oriented software system into language-level objects and to show that newly available compilation technology makes this proposal attractive with respect to performance (memory footprint and execution time). The trick for the compiler is to infer which classes are not needed for a specific configuration and to optimize the generated code accordingly. This approach opens the possibility of leveraging the good modeling capabilities of object-oriented languages to deal with fully dynamic software configuration, while being able to produce a space- and time-efficient executable when the program contains enough static configuration information.

We have demonstrated the interest of this idea on a case study from the telecommunications domain. In the most favorable cases, the GNU SmallEiffel compiler was able to infer the type of the receiver in more than 99% of the cases, and thus to optimize out the dynamic binding. Even if performance improvements resulting from type inferences look especially good in this example (due to the fine grain of its variants), we found it worthwhile to start applying this idea to all our new projects. Still, reifying variants of configuration items cannot become a widely accepted practice in software configuration management until more mainstream compilers incorporate type inference technologies. This actually seems to be work in progress for several compilers for C++ and Java.

ACKNOWLEDGMENTS

We would like to thank Dominique Colnet, the author of the SmallEiffel compiler, for the many insights he gave us on the inner working of his compiler. We also thank Chantal Brohier (Newbridge Networks) who gave us feedback on this article from an industry point of view, as well as Walter Tichy (TOSEM Associate Editor) and the anonymous reviewers.

REFERENCES

AGESEN, O. 1996. Concrete type inference: Delivering object-oriented applications. Tech. Rep. SMLI TR-96-52. Sun Microsystems, Inc., Mountain View, CA.

BENDIX, L. 1992. Automatic configuration management in a general object-based environment. In *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering* (Capri, Italy, June). 186–193.

ESTUBLIER, J. AND CASALLAS, R. 1995. Three dimensional versioning. In *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, J. Estublier, Ed. Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, Vienna, Austria, 118–135.

GALLAGHER, K. B. AND BERMAN, L. I. 1993. Applying metric-based object-oriented process modeling techniques to configuration management. In *Proceedings of the 4th International Workshop on Software Configuration Management* (Baltimore, MD, May). 79–101.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

JÉZÉQUEL, J.-M. 1996. *Object Oriented Software Engineering with Eiffel*. Addison-Wesley, Reading, MA.

JÉZÉQUEL, J.-M. 1998. Object-oriented design of real-time telecom systems. In *Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (ISORC '98, Kyoto, Japan, Apr.). IEEE Press, Piscataway, NJ.

LEBLANG, D. B. 1994. The CM challenge: Configuration management that works. In *Configuration Management*, W. F. Tichy, Ed. Trends in Software, vol. 2. John Wiley & Sons, Inc., New York, NY, 1–38.

MEYER, B. 1992. *Eiffel: The Language*. Prentice-Hall Object-Oriented Series. Prentice-Hall, Inc., Upper Saddle River, NJ.

Ray, R. J. 1995. Experiences with a script-based software configuration management system. In *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, J. Estublier, Ed. Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, Vienna, Austria, 282–287.

Tichy, W., Ed. 1994. *Configuration Management*. John Wiley and Sons Ltd., Chichester, UK.

Zendra, O., Colnet, D., and Collin, S. 1997. The SmallEiffel compiler. *SIGPLAN Not. 32*, 10 (Oct.).

Received: June 1998; revised: October 1998; accepted: November 1998