Reifying configuration management for object-oriented software

Jean-Marc Jézéquel

Reifying Configuration Management for Object-Oriented Software

J.-M. Jézéquel
Irisa/CNRS
Campus de Beaulieu
F-35042 Rennes Cedex, FRANCE
Tel: +33 299 847 192 — Fax: +33 299 847 171
E-mail: jezequel@irisa.fr

ABSTRACT
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 paper is to 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 to 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. We demonstrate this idea on an artificial case study intended to be representative of a properly designed OO software. All the performance figures we get are obtained with freely available software, and, since the source code of our case study is also freely available, they are easily reproducible and checkable.

1 INTRODUCTION

Using a solid Software Configuration Management (SCM) [18, 31] is a basic requirement in the Software Engineering Institute (SEI) capability maturity model (CMM). There are however a number of different interpretations on the exact meaning of *Software Configuration Management*. In this paper, we focus its scope to be the management of software development projects with respect to the three dimensions identified in [9]:

- targeting environmental differences (e.g., multiple platforms)
- supporting multiple versions, and controlling the status of code
- multiple developers working on the same code at the same time

Following a terminology widely adopted in the Software Engineering community [20], *variants* of configuration items are different implementations that remain valid at a given instant in time, created to handle environmental differences (logical versioning). *Revisions* are the steps a configuration item goes through over time (historical versioning), whether to handle new features, fix bugs or to support permanent changes to the environment (e.g., operating system upgrades, if the old one is no longer supported). Variants and revisions provide a two-dimensional view into the repository, with variants incrementing along one axis as required and revisions incrementing through time on the other. Versions of configuration items are understood by the SCM community to be synonymous with either revisions or variants [32]. Therefore a version of a single configuration item denotes an entry in the two-dimensional view of the repository reached from an origin through some path of revisions and variants. A third dimension is brought in when concurrent development activities are enabled (cooperative versioning): at a given point in time, concurrent activities may have a cooperative version of the same object [9]. Since many developers may be authorized to modify the same version at the same moment, each of them is in fact provided with a copy of the item, in much the same way as shared virtual memory pages can be updated using weak-consistency algorithms in distributed systems.

Even with the help of sophisticated tools [19, 23], handling these three dimensions of SCM can be a daunting (and costly) task for many projects [1, 26].

The contribution of this paper is to propose a method to simplify SCM by reifying the *variants* of 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) by in-

ferring which classes are needed for a specific configuration and optimizing the generated code accordingly. There have already been some attempts to mix the OO paradigm with the SCM problematic. Most of these attempts were trying to implement a classical SCM with the help of the OO technology [4, 6, 11, 24, 27]. Our approach is quite orthogonal, because it consists in altering the (object-oriented) design in such a way that some aspects of the SCM (the variability dimension) are vastly simplified.

This paper is organized as follows. In Section 2 we describe the factors that lead to a growing apparition of software variants, and how this variability has traditionally been addressed by more and more complex technics and tools. In Section 3 we introduce a case study and show how things can rapidly get out of hand. We then propose to reify the variability of software by eliminating the variant dimension from the three-dimensional view of the software baseline repository. Creational Design Patterns can then be used to provide the necessary flexibility for describing and selecting relevant configurations within the implementation object-oriented language, and thus benefitting from a better security implied by static typing checked by the compiler. SCM could then be implemented with much simpler tools (less costly), because only revisions would need to be dealt with. Alternatively, it could make full featured tools easier to use, thus attacking one of the perceived drawbacks of off-the-shelf SCM tools, i.e. their difficult learning curve [1, 8]. In Section 4 we discuss how new compilation technology, based on type inference, makes this proposal attractive by allowing the generation of code specialized for each variant of the software. We present performance results (memory footprint and execution time) of this approach on various systems.

In Section 5 we discuss the interests, limitations and drawbacks of our approach, as well as related works. We conclude on the perspective open by our approach.

2 SOFTWARE CONFIGURATION MANAGEMENT

2.1 Variants in Software Systems

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. We can classify these environmental differences in the following categories:

- Hardware level: most software systems must be able to drive various variants of hardware devices, e.g., multimedia or network interface boards.
- Heterogeneous distributed systems: more and more applications (singularly in the real time domain) are implemented on distributed systems made of more than one processor type, and have thus to handle such things as task allocation and functionality distribution, and eventually differences in binary formats.
- Specificities in the target operating systems: some system calls have syntax and/or semantics peculiar to a specific OS. Even more complicated are the cases when seemingly close abstractions (e.g., I/O handles in Win32 and file descriptors in Unix) must in fact be dealt with considerable differences in programming style (Win32 pro-active I/O vs. Unix reactive I/O). Note that functions whose names exist in only a subset of the supported systems cannot be linked in with a general purpose version configurable at run time.
- Compiler differences, or poorly standardized languages getting different interpretations in different compilers.
- Range of products: often, for marketing reasons, it is useful to be able to propose a range of products instead of a single suit-them-all product. For instance, one approach is to make various levels of functionality available: Demo version, Shareware, Retail, Premium, etc. Also in this category are the variants developed specifically for an important client.
- User preferences for Graphical User Interface (GUI): look-and-feel, etc.
- Internationalization: dealing with various languages and way of handling country specificities such as date and time formats, money representation etc.

Managing all the combinations between these variability factors can soon become a nightmare. Consider the case of the software for a medium sized switch in the telecommunication domain, like the Alcatel E-10. Its source code size is in the order of the million lines. Due to the many versions of the switch tailored to fit each country specificities, its configuration software also reaches the million lines range.

2.2 Traditional Solutions

One of the most primitive "solution" to these problems was to patch the executable program at installation time to take into account some variants. One of the most striking example was the word processor Wordstar under the CP/M operating system, cited in [13]. To cope with the widely different characteristics of printers and CRT terminals on CP/M systems, in addition to accommodating individual user preferences, this program

came with a configuration tool and scripts. Running the configuration tool modified configuration data in the executable image of the word processing program. Various scripts provided consistent sets of answers, corresponding to common configurations, to questions asked by the installation tool.

Device drivers are one example of configurability common to almost all operating systems. The actual binding can take place in source code, at link time, at boot time, or on demand at run time (with kernel loadable modules as in the Win32, Linux or Solaris OS).

A widely used technique for making small real time programs configurable is the static configuration table. Data structures are provided for things that might differ in different installations of the program, and the installer is responsible for providing appropriately initialized instances for a specific installation. Sometimes configuration records are not directly prepared as initialized records in the programming language of the system, but rather are produced as database entries or expressed as sentences in a grammar, with some tool provided to generate from these the programming language records the system will actually use. This can be particularly useful when several programs need to be implemented for the same configurations. Static configuration tables are not entirely satisfactory. Rarely is there provision for error checking; indeed, because they are purely declarative with no language-defined semantics, constraint verification and consistency checking can be difficult, let alone error checking. There is an implicit assumption of an associated library, where variant units of code are kept, yet there is no assistance in managing or manipulating that library.

For larger systems, one of the most popular approach consist in using conditional compilation (or assembly), implemented with e.g., a pre-processor. C programmers are familiar with the `cpp` tool, actually invoked as a first pass of the C compiler, that allows such conditional code to be written. Despite the help of sophisticated tools (such as the GNU autoconfig), this kind of code can rapidly become difficult to maintain [26]. For example, to add support for a new OS, one needs to review *all* the already written code looking for relevant #ifdef parts.

2.3 Using SCM Tools

Traditionally, SCM is implemented with check-in/checkout control of sources (and sometimes binaries) and the ability to perform builds (or compiles) of the end products. Other functions, such as *Process Management*, i.e. the control of the software development activities, will not be considered here.

Modern SCM tools have evolved from academic prototypes to full strength industrial products. Most of them now keep track of all the changes to files in secure, distributed repositories. They also support parallel development by enabling easy branching and merging. They provide version control of not only source code, but also binaries, executables, documentation, test suites, libraries, etc. Examples of such tools are Adele [5] or ClearCase [3].

For instance, ClearCase provides each developer with multiple consistent, flexible, and reproducible workspaces. It uses *Views* (similar to the views concept in databases) to present the appropriate versions of each file and directory for the specific task at hand. Views are defined by configuration specifications consisting of a few general rules. Rules may be as simple as "*the versions that were used in the last release*" or can be more complex when particular sets of bug fixes and features need to be combined. Views are dynamic — they are continually updated by re-evaluating the rules that define it. Newly created versions can thus be incorporated into a view automatically and instantly. Views allow team members to strike a balance between shared work and isolation from destabilizing changes.

The main drawbacks of these sophisticated tools is that they are very costly to use, and have a steep learning curve. Furthermore, even when these two problems are overcome, it is a matter of facts that their underlying 3-dimensional model of the repository does not provide an easy framework to mentally handle the complexity of large software developments [1, 8]. For all these reasons, their use is still not as pervasive as it could be.

3 CASE STUDY

3.1 The Mercure Software

To present the interest of a contribution to the software engineering field, people use to rely on a real software case study. However this approach has several drawbacks. The usual proprietary nature of the studied system makes it impossible for the author to give free access to all the source code and its compilation/execution environment. The community thus cannot check the validity of the study. Further, it is hard to reproduce the results since in a typical article, one lacks space to make all the context available to the reader. This lack of reproducibility defeats the scientific method, and the results are often merely empirical.

So, instead of presenting the actual application that gave rise to the ideas described in this paper [15, 16], we build a *model* (called Mercure) of this kind of software, with all of its context (including full source code and reference to a freely available compiler) made available for fetching and checking by the community. It is basically an over-simplification of this kind of soft-

See *http://www.irisa.fr/pampa/EPEE/SCM*

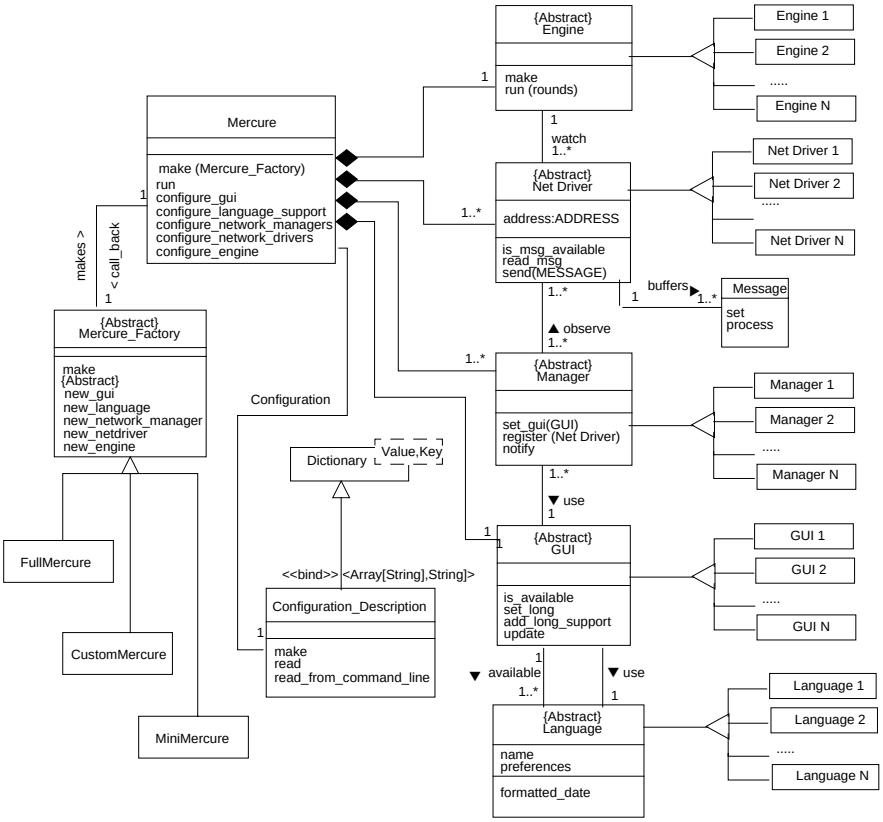

Figure 1: Class Diagram Modeling the Mercure Software in UML

ware, where only configuration management related issues would have been kept. The model is meant to be representative of the SCM issues explored in this paper, while being built in such a way that meaningful performance results can be obtained. The reader does not have to *believe* us on our good faith: she can check the relevance of the model to her own concerns to decide whether our conclusions apply to her specific case.

Mercure is a model of a communication software sending, receiving and relaying "messages" from a set of network interfaces connected to the distributed memory parallel computer (i.e. set of loosely coupled CPU) on which it runs. Mercure must handle the following variability factors (as defined in 2.1):

- Hardware level: Mercure must support a wide range of network interface boards (e.g., for ATM, various Ethernet, FDDI, ISDN, X25, etc.) from various manufacturers. Let's call V_i the number of supported boards. Since new hardware continually pops up, it must also be easy to add support for it in future releases of Mercure.

- Heterogeneous distributed systems: Mercure is to be run on such a system, thus provision must be made to deal with heterogeneous code generation and task distribution: some processors are specialized for relaying messages (switching), others for computing routes, others for network management (billing, accounting, configuring, etc.), and still others for dealing with persistent databases. Let's call V_p the considered number of specialized processors.

- Range of products: various levels (V_n) of functionality must be provided in the domain of network management.

- User preferences for GUIs: various (V_g) look-and-feel must be available.

- Internationalization: support for V_l languages must be available.

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, V_l=24 gives more than several trillions possible variants (43,980,465,111,040 to be precise).

3.2 Object Oriented Modeling of Variants

Using an object-oriented analysis and design approach, it is natural to model the commonalities between the variants of Mercure in an abstract way, and expressing the differences in concrete subclasses. Consider for example the case of the network interface boards. Whatever the actual interface, 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, that is independently of the configuration (e.g., on which kind of interface board do we actually read a message).

Dealing with multiple variants is thus moved from the implementation realm (where it is usually handled by means of conditional compilation and complex CM tools) to the problem domain (analysis and design realm), meaning that it can fully be handled within the semantics of the (OO) implementation language. This way, it can be subject to both compiler verifications and semantics-based *safe* optimizations.

In the past indeed, handling this kind of issues in an object-oriented way had a major drawback for many applications: performances. Since the choice of the proper method to call would have to be delayed until run time, we had to pay the price overhead of this dynamic binding. And this overhead could be prohibitive for some real time or performance driven applications, e.g., with Smalltalk where the inheritance hierarchy had to be search or even with C++ where the handling of dynamic binding through a *vtable* used to provoke cache misses. Fortunately, object-oriented compiler technology has made tremendous progresses in the last few years, as explained in the next section.

Figure 1 presents a class diagram of the Mercure software using the UML (Unified Modeling Language) object-model notation [28]. A Mercure system is an instance of the class of Mercure, aggregating:

- 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.
- a collection of Managers that represent the range of functionalities available,
- a collection of NetDrivers that encapsulate the network interfaces of this instance of Mercure,
- an Engine that encapsulates the actual work that Mercure has to do with its NetDrivers on a particular processor of the target distributed system.

3.3 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 a given configuration. However some care has to be taken for programming the creation of these objects to ensure that the design is flexible enough. A good approach is to use the *Creational Patterns* proposed in [12]. In our simple case, we use an *Abstract Factory* (called Mercure_Factory) to define an interface for creating Mercure variants. The class Mercure_Factory features one *Factory Method* (encapsulating the procedure for creating an object) for each of our 5 variability factors. The Factory Methods are parameterized to let them create various kinds of products (i.e. variants of a type), depending on the dynamic Mercure configuration selected at runtime. These Factory Methods are abstractly defined in the class Mercure_Factory, and given concrete implementations in its subclasses, 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).

Building an actual variant of the Mercure software then consists in implementing the relevant concrete factory. By restricting at compile time (that is 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 she wants. Since this is done at compile time, it should be possible to generate an executable code *specialized* towards the selected Mercure

On this diagram, only a rather flat inheritance hierarchy is suggested, but it is evident that the designer should factorize the commonalities between subclasses in an inheritance graph as deep as required.

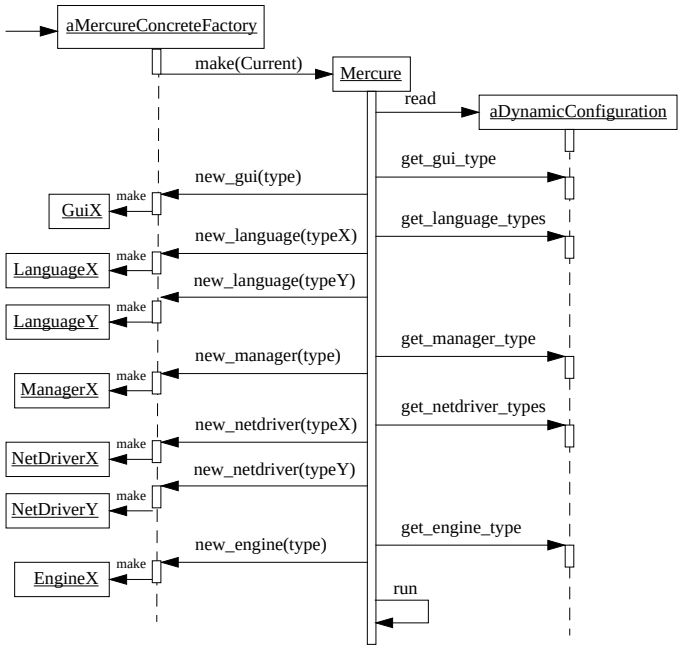

Figure 2: Dynamic Configuration of the Mercure Software (UML Sequence Diagram)

variant. In the next section, we show how this can be done automatically with current compiler technology.

4 COMPILATION TECHNOLOGY AND PERFORMANCE RESULTS

4.1 Principle of Type Inference and Code Specialization

Good object-oriented programming relies on dynamic binding for structuring a program flow of control —OO programming has even been nicknamed "case-less programming". Most of the time, a routine (a method call) applies to a given object, called *target* or *receiver*. Dynamic binding allows the choice of the actual version of the routine to be delayed until run time: the exact type (called the dynamic type) of the receiver need not be known at compile time. Whenever more than one version of a routine might be applicable, it ensures that the most directly adapted to the target object is selected. In statically typed languages (e.g., C++, Eiffel, Java, Ada95), a receiver's type must be declared beforehand (this is called the receiver's static type). Then the receiver's dynamic type must be a subtype of its static type.

In this context, the main goal of the compilation techniques based on *type inference* consists in *statically* computing the set of types a *receiver* may assume at a given point in a program. In the most favorable case, this set is a singleton and thus the routine 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 routines 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 receiver (corresponding to the C++ RTTI) to select the relevant procedure 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 SmallEiffel [7], a free Eiffel compiler distributed under the terms of the GNU General Public License as published by the Free Software Foundation So we have implemented the Mercure software with Eiffel, and used the SmallEiffel compiler to make a number of measures. Eiffel [21] 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++, Ada95 or Java.

4.2 Experimental Conditions

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, including all the configurable parts. That means that *anyone* of the trillions of combinations can be dynamically chosen at runtime: all calls to the variant methods must be dynamically bound.

CustomMercure This restricted version of the program only includes support for 8 different network drivers, 5 different languages and 5 different processor types. Only one network manager, and one GUI are available, thus allowing some method calls to be statically bound.

MiniMercure A minimal version of the software, with only one of each configurable part available: support for ENGINE1, GUI2, LANGUAGE3, MANAGER4 and NETDRIVER5 only. This limited support would theoretically allow every method to be statically bound, and thus the resulting code could have the same structure as with e.g., the #ifdef based pre-processor method.

These three variants use exactly the same software baseline. The only difference is that a different Mercure

SmallEiffel can be downloaded from *ftp://ftp.loria.fr/pub/loria/genielog/SmallEiffel.*

Version	Eiffel LOC (config.)	Eiffel LOC (user)	Eiffel LOC (total)	Type inference Score	C LOC (generated)
FullMercure	96	2903	10056	93.29%	7135
CustomMercure	60	1421	8574	96.79%	3839
MiniMercure	36	713	7866	99.29%	2639
HelloWorld	–	6	4478	100.00%	189

Table 1: Compile time statistics

concrete factory is selected as the root class, that is the class containing the entry point of the application (See Figure 1).

4.3 Compile Time Statistics

In this section, we compare compile time statistics for the various variants of the Mercure software with respect to the minimal "Hello, world!" program (see Table 1). We display the number of Eiffel lines of code (LOC) for describing the configuration (i.e. the number of LOC of the relevant Mercure concrete factory), the number of LOC written by the programmer, as well as the total number of lines in all the classes needed by the application (including the libraries).

Then comes the type inference score, that is the ratio of dynamic calls that could be replaced by direct call at compile time. It ranges from 93% to more than 99%. This means that the SmallEiffel compiler (version -0.87) has been able to early bind most of the (conceptually) dynamic binding in the MiniMercure version.

Finally, the size of the generated C code is shown. Note that it includes the SmallEiffel runtime system (whose size may be approximated by the "Hello, world!" one). The small size of the code generated for the MiniMercure version illustrates the ability of the SmallEiffel compiler to take advantage of its knowledge of the living types to efficiently specialize generated C code: only code relevant to the specific variant of the Mercure software is actually generated.

4.4 Memory Footprint and Runtime Performances

All versions have exactly the same dynamic behavior, because the dynamic configuration we choose for the Mercure and CustomMercure variants is the one selected at compile time in MiniMercure (i.e., we give the configuration as a set of command line parameters: `-run 10000 -engine 1 -gui 2 -lang 3 -manager 4 -netdriver '5 5 5 5 5 5 5 5'`). Their output is thus exactly the same.

The results presented in Tables 2 and 3 have been made on a PC486 system running Linux 1.2.13 (and GCC 2.7.0, optimization level -O3) and on a Sparc running Solaris 5.0 (and GCC 2.7.2.1, optimization level -O3).

Version	Footprint	Run Time	Speed-up
FullMercure	82544	1.319	0%
CustomMercure	40512	1.159	12.1304%
MiniMercure	26880	1.086	17.6649%

Table 2: Run time statistics on Linux

Version	Footprint	Run Time	Speed-up
FullMercure	116152	0.486	0%
CustomMercure	57696	0.433	10.9053%
MiniMercure	38824	0.413	15.0206%

Table 3: Run time statistics on Sparc/Solaris

Note that because all three variants have the same dynamic behavior (they do exactly the same thing), their use of dynamic memory is also identical. Despite the system being designed for a fully dynamic configuration, the compiler is able to use type inference to detect what is in fact configured statically in specialized versions of Mercure factories to generate code nearly as compact and efficient as if it had been written statically from the beginning. In the MiniMercure case, the generated code has the same structure as the one that would have been obtained with e.g., the #ifdef based pre-processor method. The performance differences between MiniMercure and FullMercure represent the maximum price that the designer would have to pay for trading time and space performances for dynamic configuration capabilities. But what is much more interesting is that with *exactly the same software baseline*, the designer can easily choose his own trade-off between these two properties: he has just to select the relevant concrete factory.

5 DISCUSSION AND RELATED WORK

5.1 Discussion

Our approach is not the ultimate solution to all SCM problems. It has a number of drawbacks and advantages:

- It forces some SCM issues (variant management) to be dealt with during the design phase of the software. But according to us, it belongs there, because it makes the notion of a product family much more concrete. There is one concrete factory for each variant of the product, and no more need to understand the variations between variants in terms of "diff" listings.
- A compiler is able to do type inference only if it has 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 the overall edit/compile/link/test time. With respect to medium size programs, for example it only takes 10 seconds on a Pentium Pro 200 to have SmallEiffel compile itself (50kLOC).
  - source protection: having access to the full code does not mean full *source* code, because the source can be pre-compiled in a "distributable" format, e.g., Java *.class* formats or Eiffel "pre-compiled" 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, see below) should be enough for many sites.
- Programming the concrete factories to specify the configuration is straightforward, but quite tedious. This could easily be generated by e.g., a simple Tcl/Tk shell. This shell would also encapsulate the call to the compiler and thus could be able to retrieve the name of all the files used in the compilation. Using this information, a snapshot of the full configuration (including the compiler, linker etc.) could be assigned a symbolic version name and stored in a repository (e.g., using RCS).
- 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 (e.g., the various existing Module Interconnection Languages, as in Adele [5], Proteus [10], etc.)

5.2 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). Thus the Partial Evaluation community trick known as *The Trick* (see [17]) can be applied to specialize the general program 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. Ole Agesen's recent PhD thesis [2] contains a complete survey of related work. Reviewed systems range from purely theoretical ones [33] to systems in regular use by a large community [22], via partially implemented systems [29, 30] and systems implemented on small languages [14, 25].

Related work from the SCM point of view have already been extensively discussed all along this paper. Here we restrict ourselves 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 [6, 11]. 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.

6 CONCLUSION

Our contribution in this paper was to propose a method to simplify software configuration management by reifying the *variants* of 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) by inferring which classes are needed for a specific configuration and optimizing the generated code accordingly. This approach opens the possibility of leveraging the good modeling capabilities of OOL to deal with fully dynamic software configuration, while being able to produce space and time efficient executable when the program contains enough static configuration information. We have illustrated this idea with a small case study representative of a properly designed OO software. All the performance figures we get are obtained with freely available software, and, since the source code of our case study is also freely available, they are easily reproducible and checkable.

In the most favorable cases, the SmallEiffel compiler is able to infer the type of the receiver in up to 100% of the cases, and thus to optimize out the dynamic binding. We believe that this approach can become mainstream when commercial compilers incorporate these kinds of technologies. From advertisement flyers we have seen, this seems to be work in progress for several compilers for C++ and Java.

ACKNOWLEDGEMENTS

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 paper from an industry point of view.

REFERENCES

[1] C. Adams. Why can't i buy an scm tool. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in Lecture Notes in Computer Science, pages 278–281. Springer-Verlag, Oct. 1995.

[2] O. Agesen. *Concrete Type Inference : Delivering Object-Oriented Applications*. PhD thesis, Department of Computer Science of Standford University, Published by Sun Microsystem Laboratories (SMLI TR-96-52), 1996.

[3] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner. ClearCase MultiSite: Supporting geographically-distributed software development. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in Lecture Notes in Computer Science, pages 194–214. Springer-Verlag, Oct. 1995.

[4] V. Ambriola and L. Bendix. Object-oriented configuration control. In *Proceedings of the 2nd International Workshop on Software Configuration Management*, pages 133–136, Princeton, New Jersey, Oct. 1989.

[5] N. Belkhatir and W. L. Melo. Supporting software development processes in Adele 2. *The Computer Journal*, 37(7):621–628, May 1994.

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

[7] S. Collin, D. Colnet, and O. Zendra. Type Inference for Late Binding: The SmallEiffel Compiler. In *Joint Modular Languages Conference*, pages 67–81. Lecture Notes in Computer Science, Springer-Verlag, 1997.

[8] S. Dart and J. Krasnov. Experiences in risk mitigation with configuration management. Technical report, Continuus Software Corporation, 108 Pacifica, Irvine, California, 1995. Delivered at 4th SEI Risk Conference, November 1995.

[9] J. Estublier and R. Casallas. Three dimensional versioning. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in Lecture Notes in Computer Science, pages 118–135. Springer-Verlag, Oct. 1995.

[10] J. Floch. Supporting Evolution and Maintenance by using a Flexible Automatic Code Generator. In *Proceedings of the 17th International Conference on Software Engineering*, pages 211–219, Apr. 1995.

[11] K. B. Gallagher and L. I. Berman. Applying metric-based object-oriented process modeling techniques to configuration management. In *Proceedings of the 4th International Workshop on Software Configuration Management (Preprint)*, pages 79–101, Baltimore, Maryland, May 1993.

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

[13] W. M. Gentleman. Managing configurability in multi-installation realtime programs. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, pages 823–827, Nov. 1989.

[14] J. Graver and R. Johnson. A Type System for Smalltalk. In *Proceedings of POPL*, pages 139–150, 1990.

[15] J.-M. Jézéquel. *Object Oriented Software Engineering with Eiffel*. Addison-Wesley, Mar. 1996. ISBN 1-201-63381-7.

[16] J.-M. Jézéquel. Object-oriented design of real-time telecom systems. In *IEEE International Symposium on Object-oriented Real-time distributed Computing, ISORC'98, Kyoto, Japan*, Apr. 1998.

[17] N. D. Jones. Partial evaluation, self-application, and types. In M. S. Paterson, editor, *17th International Colloquium on Automata, Languages, and Programming (ICALP), Warwick, England*, Lecture Notes in Computer Science, pages 639–659. Springer-Verlag, New York, N.Y., July 1990.

[18] D. Leblang and P. H. Levine. Software configuration management: Why is it needed and what should it do? In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in Lecture Notes in Computer Science, pages 53–60. Springer-Verlag, Oct. 1995.

[19] D. B. Leblang. The CM challenge: Configuration management that works. In W. Tichy, editor, *Configuration Management*, pages 1–38. John Wiley and Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 1994.

[20] S. A. MacKay. The state-of-the-art in concurrent, distributed configuration management. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in Lecture Notes in Computer Science, pages 180–193. Springer-Verlag, Oct. 1995.

[21] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[22] R. Milner. A Theory of Type Polymorphism in Programming. In *Journal of Computer and System Sciences*, pages 348–375, 1978.

[23] V. Mosley, F. Brewer, R. Heacock, P. Johnson, G. LaBarre, V. Maz, and T. Smith. Software configuration management tools: Getting bigger, better, and bolder. *Crosstalk: The Journal of Defense Software Engineering*, 9(1):6–10, Jan. 1996.

[24] F. Oquendo. Acquiring experiences with object-based and process-centered CASE engironment architectures for configuration management systems. In G. Forte, N. H. Madhavji, and H. A. Muller, editors, *Proceedings of the 5th International Workshop on Computer-Aided Software Engineering*, pages 14–18, Montreal, Quebec, Canada, July 1992.

[25] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of 6th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA'91)*, pages 146–161, 1991.

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

[27] H. Render and R. Campbell. An object-oriented model of software configuration management. In P. H. Feiler, editor, *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 127–139, Trondheim, Norway, June 1991.

[28] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1997.

[29] N. Suzuki. Inferring Types in Smalltalk. In *Eighth Symposium on Principles of Programming Languages*, pages 187–199, 1981.

[30] N. Suzuki and M. Terada. Creating Efficient System for Object-Oriented Languages. In *Eleventh Annual ACM Symposium on the Principles of Programming Languages*, pages 290–296, 1984.

[31] W. Tichy, editor. *Configuration Management*. John Wiley and Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 1994.

[32] W. F. Tichy. Programming-in-the-large: Past, Present, and Future. In *Proceedings of the 14th International Conference on Software Engineering*, pages 362–367, May 1992.

[33] J. VitekN, N. Horspool, and J. Uhl. Compile-Time Analysis of Object-Oriented Programs. In *International Conference on Compiler Construction*, pages 237–250. LNCS 641, Springer Verlag, 1992.