

Using Compile-Time Reflection for Objects' State Capture¹

Marc-Olivier Killijian, Juan-Carlos Ruiz-Garcia, Jean-Charles Fabre

LAAS-CNRS, 7 Avenue du Colonel Roche
31077 Toulouse cedex, France
{killijian,fabre,ruiz}@laas.fr

1 Motivations

Checkpointing is a major issue in the design and the implementation of dependable systems, especially for building fault tolerance strategies. Checkpointing a distributed application involves complex algorithms to ensure the consistency of the distributed recovery state of the application. All these algorithms make the assumption that the internal state of the application objects can be obtained easily. However, this is a strong assumption and, in practice, it is not so easy when complex active objects are considered. This is the problem we focus on in our current work. The available solutions to this problem either rely on :

- a specific hardware/operating-system/middleware that can provide runtime information about the mapping used between memory and application objects;
- a reflective runtime which provides access to class information, such as Java Serialization [1] based on the Java Reflection API;
- intervention at the language level, rewriting the original code to add checkpointing mechanisms (e.g. Porch [2] for C programs);
- the provision of state handling mechanisms by the user; since the implementation of these mechanisms must be error-free, this is a weak solution.

When neither the underlying system nor the language runtime is sufficiently reflective to provide type information about the object classes, the only solution is to obtain this information at the language level. Two approaches can be investigated here: (i) developing a specific compiler (as Porch does) or (ii) using a reflective compiler such as OpenC++ [3] or OpenJava [4]. Since diving into a compiler is very complex and can lead to the introduction of new software faults, we think that the latter is a better solution. Clearly, compile-time reflection opens up the compilation process without impacting the compiler itself. Off-the-shelf compilers can thus be used to produce the runtime entities.

¹ This work has been partially supported by the European Esprit Project n° 20072, DEVA, by a contract with FRANCE TELECOM (ref. ST.CNET/DTL/ASR/97049/DT) and by a grant from CNRS (National Center for Scientific Research in France) in the framework of international agreements between CNRS and JSPS (Japan Society for the Promotion of Science).

2 Approach Overview

To provide state handling facilities to target objects, we have to implement two new methods for each class: `saveState` and `restoreState`. The former is responsible for saving the state of an object and the latter for restoring this state; these methods are similar to the `readObject` and `writeObject` methods provided by the Java runtime.

These methods must save/restore the values of each attribute: basic and structured types of the language, arrays, and inherited fields. Furthermore, they must be able to traverse object hierarchies, i.e. save/restore the objects included in the object currently processed. To generate these methods we use OpenC++, an open-compiler for C++ which provides a nice API for both introspection and intercession of application classes. Thanks to the static type information it provides, we are able to analyse the structure of the classes and can thus generate the necessary methods for checkpointing.

However, since C++ is an hybrid between object-oriented and procedural languages, some of its features are very difficult to handle. Features which break the encapsulation principle must be avoided from a dependability viewpoint: friend classes or functions and the pointer model of C++. We decided to handle only a single level of indirection for pointers and to forbid the use of pointer arithmetic; pointers can thus be seen as simple object references as in Java.

Filtering programs and rejecting those that don't respect these restrictions is easy to implement using compile-time reflection. Enforcing other programming restrictions is also useful for validation aspects. Dependability can be significantly enhanced by restricting to a subset of a language [5] in many respects.

It is worth noting that this approach does not deal with multithreaded objects. Indeed, compile-time reflection is not sufficient to save the state of threads, thread introspection is needed [6].

3 Current Implementation Status

We have implemented this approach using OpenC++ 2.5.1 and GCC 2.8.1 on both Solaris and Linux operating systems. Firstly the application program is parsed by the metaclasses we designed, these metaclasses generate the necessary facilities (save/restore state methods and related state buffer classes). Secondly, the C++ compiler compiles the instrumented code.

We are currently working on the validation of this toolkit. We are using some application benchmarks in order to evaluate the correctness of the states obtained (the coverage of the class' structure analysis), and to evaluate the efficiency of the state capture/restoration compared to Java serialization and Porch checkpointing. The first efficiency results obtained on simple examples are promising.

We have also implemented an optimisation of this approach that saves only the modified attributes of an object. This technique uses runtime reflection in order to know which of the object's members have been modified since the last checkpoint.

The idea is to save/restore a delta-state instead of the full object state. Since an object's methods often modify only a small subset of its attributes, this approach is more efficient in practice. It is worth noting that this new technique can only be applied for checkpointing. Cloning an object requires its full state to be available.

4 Conclusion

The work briefly describe here is part of a reflective architecture for dependable CORBA applications [7] which implements fault-tolerance mechanisms as metaobjects on any off-the-shelf ORB. The metaobject protocol controls both object interaction and object state. It has been developed for C++ objects and is currently being ported for Java objects using OpenJava. The implementation relies on a combined use of tools and techniques, such as open compiler, IDL compiler and reflective runtime when available.

Acknowledgements. We would like to thank Shigeru Chiba and Michiaki Tatsubori for their contribution to this work and invaluable assistance in using OpenC++.

References

- [1] Sun Microsystems, "Java Object Serialization Specification".
- [2] V. Strumpen and B. Ramkumar, "Portable Checkpointing for Heterogeneous Architectures," in *Fault-Tolerant Parallel and Distributed Systems*, D. Avresky, R. and D. Keli, R., Eds.: Kluwer Academic Press, pp. 73-92, 1998.
- [3] S. Chiba, "A Metaobject Protocol for C++," presented at OOPSLA, Austin, Texas, USA, pp. 285-299, 1995.
- [4] M. Tatsubori, "An Extensible Mechanism for the Java Language", Master of Engineering Dissertation, Graduate School of Engineering, University of Tsukuba, University of Tsukuba, Ibaraki, Japan, Feb 2, 1999.
- [5] P. J. Plauger, "Embedded C++", appeared in *C/C++ Users Journal*, vol.15, issue 2, February 1997.
- [6] M. Kasbekar, C. Narayanan, and C. R. Dar, "Using Reflection for Checkpointing Concurrent Object Oriented Programs" Center for Computational Physics, University of Tsukuba, UTCCP 98-4, ISSN 1344-3135, October 1998.
- [7] M.-O. Killijian, J.-C. Fabre, J.-C. Ruiz-Garcia, and S. Chiba, "A Metaobject Protocol for Fault-Tolerant CORBA Applications," presented at IEEE Symposium on Reliable Distributed Systems, West Lafayette, Indiana, USA, pp. 127-134, 1998.