

Experiences in Adopting Real-Time Java for Flight-like Software

Edward G. Benowitz, Albert F. Niessner
 Jet Propulsion Laboratory, California Institute of Technology
 4800 Oak Grove Drive
 Pasadena, CA 91109
 {Edward.G.Benowitz, Albert.F.Niessner}@jpl.nasa.gov

Abstract—This work involves developing representative mission-critical spacecraft software using the Real-Time Specification for Java(RTSJ)[1]. Utilizing a real mission design, this work leverages the original flight code from NASA's Deep Space 1(DS1), which flew in 1998. However, instead of performing a line-by-line port, the code is re-architected in pure Java™, using best practices in Object-Oriented(OO) design. We have successfully demonstrated a portion of the spacecraft attitude control and fault protection, running on a standard Java platform, and are currently in the process of taking advantage of the features provided by the RTSJ. In particular, we are focusing on interaction with RTSJ's scheduling and memory area frameworks. Our goal is to run on flight-like hardware, in closed-loop with the original spacecraft dynamics simulation.

In re-designing the software from the original C code, we have adopted OO techniques for flight software development. Specifically, we have taken advantage of design patterns[7], and have seen a strong mapping from certain patterns to the flight software. To ensure the correctness of measurement units, numerical computations are performed via an abstraction layer that checks measurement units at compile-time.

Our approach places an emphasis on pluggable technology. Interfaces, in conjunction with a façade pattern, expose only the behavior of a subsystem, rather than exposing its implementation details. Since the RTSJ reference implementation does not currently support debugging, we chose to apply pluggable technology to the scheduler and memory allocation interfaces. Thus, real-time client code can be run on a standard Java virtual machine, allowing the code to be debugged in a graphical development environment on a desktop PC at the cost of decreased real-time performance. Once non-real-time issues have been debugged, the real-time aspects can be debugged in isolation on an RTSJ-compliant virtual machine.

I. INTRODUCTION

Java introduces a number of advantages for software development by improving software maintainability. We seek to take advantage of Java's features for flight software development. The RTSJ is a relatively new extension to Java which introduces real-time features. Since the specification is a new technology, there are few examples of real-time Java being applied to non-trivial problems. Our work involves creating real-time Java software which implements a realistic spacecraft control loop. In this section, we discuss the motivation for our work, the advantages of Java, and our approach.

A. Motivation

Flight software has a high development cost, due in part to the difficulty in maintaining the code. The lack of maintain-

ability stems from the limitations of current implementation languages (typically C or C++). There is a lack of strong type-checking and parameter checking. Memory can easily be corrupted due to the lack of pointer checking and array-bounds checking. Without operating system protections, these problems can occur as silent failures. Concurrency primitives are very low-level, and are not part of the language. A typical program will abound with error-prone switch statements and preprocessor directives. And only a globally shared namespace is available.

Pluggable components cannot be expressed with traditional flight software techniques. A pluggable component is a software article which only exposes its interface (behavior) and not its implementation. Pluggable components allow different implementations to be swapped, without requiring modifications to the rest of the code. The C language does not provide this level of encapsulation mostly because of its procedural orientation. Although C++ attempts to provide encapsulation, multiple inheritance problems exist. Additionally, the encapsulation can easily be broken by using the friend keyword.

B. Advantages of Java

To address these issues, we are investigating Java as an implementation language for flight software. Java improves maintainability with its strong type-checking at both compile-time and run-time. Additionally, Java checks array boundaries, and ensures that variables are initialized. Standard Java provides automatic memory management, and Real-Time Java allows several forms of manual memory management where required (see II-B for details). Multi-threading and higher-level concurrency primitives are built into the language as well. Java can easily express pluggable components, as the notion of an interface is an explicit part of the language. Java provides for full encapsulation, and allows single inheritance with multiple interface inheritance. Java also provides extensibility through inheritance and dynamic class-loading.

Aside from the advantages of the language itself, the Java platform includes a large standard class library with support for most programming needs. Due to the large Java developer community, additional Java components are available from the internet, often for free.

According to NIST[9], Java's higher level of abstraction leads to increased programmer productivity. The Java platform,

coupled with Java language, improves application portability. Additionally, Java is easier to master than C++, and supports component integration and reuse.

C. Approach

We specifically chose to favor maintainability above all else during the architectural, design, and implementation phases of the development. Maintainability requires making extensive use of design patterns, taking full advantage of Java language features, using pluggable technology, and making appropriate use of commercial, off-the-shelf libraries and tools. During the performance evaluation phase, it is known that some maintainability will be sacrificed in order to optimize parts of the system to meet performance requirements. However, empirical evidence from systematic profiling of the application will dictate where and what is to be optimized, as opposed to prematurely attempting to optimize based on intuition, instincts, and/or assumptions of behavior.

For our implementation approach, we have chosen two subsystems to focus on. From the attitude control system, we have implemented a real-time control loop which detumbles a simulated spacecraft. In a detumble operation, a spacecraft with some initial set of angular velocities is controlled such that its angular velocity is reduced to a value close to zero. Our pure Java detumble control loop runs in real-time on a standard Java virtual machine. In addition, we have ported a C++ fault protection subsystem to Java, demonstrating a degree of autonomous behavior.

D. Tools

COTS graphical development tools were used extensively in this project. Specifically, the open-source Eclipse[6] integrated development environment provided graphical code editing, browsing, debugging, and refactoring capabilities. Headway's Review[8] product was used to graphically inspect our design, and allowed us to maintain a consistent architecture. Additionally, JProbe[13] was used to examine memory usage, and to identify critical regions for future optimization.

An RTSJ-compliant virtual machine is required for running real-time Java applications. In addition to the the RTSJ reference implementation, several additional RTSJ implementations are just now becoming available: JRate[3], OVM[10], and FLEX[11].

E. Pluggable Components, Factories, and Dynamic Class-Loading

Pluggable components are specified by an interface because implementation details are not visible from classes using the components. To hide the implementation class of a particular component, a factory is used instead of directly calling a constructor. The factory is tasked with constructing a particular instance of the pluggable component, and returning the component as an interface. An abstract factory provides a further step of abstraction. Each instance of an abstract factory can construct an instance of a pluggable component in a different fashion, typically instantiating a different implementation class.

We have used dynamic class-loading, in conjunction with abstract factories as a replacement for the C preprocessor. By using this approach, we allow the user of an application to choose at run-time the implementation corresponding to a particular interface. That is, the implementation of a pluggable component can be chosen at run-time. Clients of the interface use an abstract factory to request an instance of an interface. A particular implementation of an interface will have its own concrete factory as a subclass of the abstract class factory. The proper concrete class factory is dynamically loaded at run-time, returning the corresponding implementation of the interface. The advantage of this approach is that we can swap out implementations at run-time. Specifically, this was used to choose between our desktop scheduler implementation and the RTSJ scheduler implementation at run-time.

In the long term, dynamic class-loading has much greater potential for spacecraft missions. Current practice requires reloading a binary image of the executable to a spacecraft, followed by a reboot. We envision that Java's dynamic class-loading facilities could be used to provide additional functionality to a spacecraft by uplinking new class files to a running system, without requiring a reboot. This capability is outside of our current scope, but would be an interesting avenue for further research.

II. REAL-TIME LAYER

There are several problems with traditional Java implementations that make it unsuitable for real-time programming, such as timing, scheduling, and memory management. The Real-Time Specification for Java addresses these issues by providing tighter semantics on scheduling, and by providing new memory management paradigms. We now discuss our wrapper layer for accessing these real-time services. We briefly introduce RTSJ's memory and scheduling features, and discuss some of the issues involved with the new memory management paradigm.

A. Scheduler

The nature of flight software requires that certain threads execute at certain times, and the times that the threads execute depends on the type of work being done. For instance, control loops run periodically while watchdogs run once at some time to signal the system of a potential problem. Defining the temporal boundaries and constraints for these threads is independent of the scheduling algorithm being used, but communicating these constraints to the scheduler is dependent on the scheduling algorithm and its implementation. We chose to apply the pluggable technology approach to our scheduler so that we can use whatever scheduling algorithm is available to us. Currently, we provide several varieties of scheduling requests:

- A one-shot timer. The scheduler will run a block of code after at a specific time.
- Periodic behavior: The scheduler will then run the block of code at the client specified rate.
- Standard: The scheduler will run the block of code when possible.

All of these behaviors can also specify a deadline that when crossed will cause a secondary block of code to be executed. Additionally we provide facilities for specifying a maximum percentage of CPU usage by a particular thread. The scheduler to be used is selected at run-time and instantiated through the use of dynamic class loading and factories.

Since the RTSJ reference implementation does not currently support debugging, our choice of pluggable technology allowed us to use the desktop for debugging. When running within an RTSJ-compliant virtual machine, our scheduler interface simply delegates out to the underlying RTSJ implementation. However, when running on a standard desktop Java virtual machine, the scheduler component uses our own implementation, written only using standard Java features. We emulate, as best as possible, the real-time scheduling features on a standard Java platform. Clients may choose between the RTSJ scheduler implementation and the desktop scheduler implementation at run-time.

Thus, real-time client code can be run on a standard Java virtual machine, allowing the code to be debugged in a graphical development environment on a desktop PC at the cost of decreased real-time performance. Once non-real-time issues have been debugged on a standard Java VM, the real-time issues can be debugged in isolation on an RTSJ-compliant virtual machine.

B. Memory Areas

With the addition of the RTSJ's scheduling and memory management features, come new failure modes and programming pitfalls. The developer must consciously avoid violating memory area rules, and must ensure that no memory leaks occur. We present a series of guidelines for using the RTSJ memory management features. We provide a set of recommendations for memory allocation, showing scenarios that take advantage of memory areas provided by RTSJ. In addition, restrictions are placed on memory allocation scenarios that are particularly error-prone.

1) *Immortal memory* : Immortal memory is a new allocation scheme provided by the RTSJ. Once an object is allocated in Immortal memory, it is never freed. The advantage of this approach is that objects allocated in immortal memory have no need for interaction with the garbage collector. The disadvantage is that memory leaks are now possible. We recommend that allocations to immortal memory be performed in static initializers. We also require that object which are running in immortal memory only allocate in their constructors. With these restrictions in place, memory leaks can be avoided. However, this also places severe restrictions on which classes may be used. To use a JDK class while running in immortal memory, one must inspect the source code to ensure that allocations are only performed in the constructor. For the purposes of our own application, Immortal memory was primarily used to load singleton factories.

2) *Scoped memory usage*: Scoped memory provides a means to dynamically allocate and free memory without using the garbage collector. Object allocated within a scope persist for the lifetime of the scope. Once the number of threads

within a scope reaches zero, all objects allocated within the scope are destroyed. Additionally, scopes may be nested. The advantage for application programmers is that a large number of objects can be allocated and freed at once, without creating excess work for the garbage collector. One can think of scopes as a generalization of the C stack with the exception that the objects are finalized in the case of scopes.

A particular scoped memory region is represented by a scoped memory object, which itself must be allocated in a memory region. If one allocated a scoped region on the heap, the scoped memory object itself would be subject to interference from the garbage collector. For our application, all threads are created at application startup time. In this case, we can allocate scopes in immortal memory, and have examined the possibility of creating separate scopes on a per-thread basis. This scope allocation paradigm is quite similar to having one C stack per thread. The thread would then enter its own scoped memory, perform allocations, and then leave the memory area, automatically destroying the scope-allocated data. The size of the scope can be determined by profiling the memory usage characteristics of a particular thread, taking into consideration the requirements of the application and the available hardware resources.

The difficulty with scopes is their limited lifetimes. We enter and leaving a scoped memory region once for every iteration of our control loop. However, some data will need to persist beyond the lifetime of the scope, so we must provide mechanisms for copying data out of a scoped memory region. To facilitate this, we recommend providing memory areas as parameters to factories. These factories could then be used to copy and construct objects in arbitrary memory areas.

III. UNITS

A. Problems with past practice

In current flight software projects, the measurement units are not explicitly part of the software. Perhaps measurement units are designated in an external document or in code comments, but there are no automated checks at either compile-time or at run-time to ensure that unit arithmetic is correct. For example, multiplying a velocity by a time should result in a distance. But since values are only represented as doubles, nothing prevents the developer from incorrectly treating the result of such an operation as a force, for example. We have already seen the disastrous consequences of incorrect units in the Mars Climate Orbiter mission.

B. Our approach

To remedy this problem, we advocate making measurement units an integral part of the application code. Our package provides compile-time checking of measurement units. We provide interfaces for physical units, such as forces, distances, and times, and allow scalars, matrices, and tensors of values with physical units. With measurement units explicitly part of our code, we gain a number of advantages. Since measurement units are checked at compile-time, bugs are detected sooner, with a lower cost to repair them. Specifically, by using the units framework in our development, the detumble control loop

was debugged in only 13 iterations. Because we knew that the measurement units were correct, pinpointing the actual cause of the errors became simpler.

In implementing our units framework, we have made use of COTS class libraries. Since units are pluggable components, alternative implementations are possible. For performing matrix and vector operations, we take advantage of the classes providing such functionality in Java3d. Additionally, for unit representation, we make use of the Jade library[4]. The admitted disadvantage of using Java for this situation is the lack of operator overloading, since the syntax for performing arithmetic does become quite verbose.

IV. CONCLUSION

A. Summary

We have developed a pure Java prototype attitude control system, capable of performing a detumble maneuver in real-time, along with a pure Java fault protection subsystem. In developing this prototype, we have applied best practices in OO design, including design patterns. We have demonstrated the applicability of pluggable components. The measurement units facility allows the checking of unit correctness at compile-time. Based on work with our units framework, we believe that operator overloading would be beneficial to real-time Java software development. We have explored the features of the RTSJ, discussing the usage of memory areas. We have created a pluggable real-time abstraction layer, enabled debugging on a standard Java platform.

V. ACKNOWLEDGMENTS

This work was supported in part by the Center for Space Mission Information and Software Systems(CSMISS) at the Jet Propulsion Laboratory, and by the Ames Research Center.

REFERENCES

- [1] G. Bollella , J. Gosling, B. Brosgol, P. Dibble, S. Furr, M. Turnbull, *The Real-Time Specification for Java*, Addison-Wesley, 2000.
- [2] J. Cooper, *The Design Patterns Java Companion*, Addison-Wesley, 1998.
- [3] A. Corsaro and D.C. Schmidt. "Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems." Technical Report 2002-001, University of California, Irvine, 2002.
- [4] J.M. Dautelle, "JADE Java Addition to Default Environment", <http://jade.dautelle.com/>, 2002.
- [5] P. Dibble, *Real-Time Java Platform Programming*, Prentice Hall, 2002.
- [6] "Eclipse.org" , <http://www.eclipse.org/>, 2003.
- [7] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.
- [8] "Headway Software", <http://www.headwaysoft.com/>, 2003.
- [9] NIST Special Publication 500-243, *Requirements for Real-time Extensions for the Java Platform: Report from the Requirements Group for Real-time Extensions for the Java Platform*, 1999.
- [10] OVM/Consortium, "OVM: An Open RTSJ Compliant JVM." <http://www.ovmj.org>, 2003.
- [11] M. Rinard et al., "FLEX Compiler Infrastructure", <http://www.flex-compiler.lcs.mit.edu/>, 2003.
- [12] N. Rouquette, T. Neilson, and G. Chen, "The 13th Technology of DS1." *Proceedings of IEEE Aerospace Conference*, 1999.
- [13] "Sitraka JProbe", <http://www.sitraka.com/software/jprobe/>, 2003.