Use of PERC Pico for Safety Critical Java

A. Marc Richard-Foy[1], B. Tobias Schoofs[2], C. Eric Jenn[3], D. Ludovic Gauthier[1], E.Kelvin Nilsen[4]

1: Atego SAS, 143 bis avenue de Verdun, 92442 Issy-les-Moulineaux cedex, France
2: GMV-Skysoft Portugal, Av. D. João II, Lote 1.17.02, Torre Fernão Magalhães -7°, 1998-025, Lisboa - Portugal
3: Thales Aerospace Division, 105, av. Eisenhower, 31036 Toulouse, France
4: Atego, 125 E. Main St., #501, American Fork, UT 84003, USA

Abstract:
The use of Java for safety-critical development appeals because Java's type system is much more rigorous than C and C++. As an object-oriented language, Java's standard libraries and certain language constructs make extensive use of temporary objects. One of the challenges with using real-time Java for safety-critical development is the difficulty of managing memory for temporary objects without relying upon a tracing garbage collector. The PERC Pico product offers a unique approach to this problem, using a combination of programmer-supplied annotations and an integrated static analyzer to guide the PERC Pico code generator to perform all temporary object allocations from the run-time stack rather than depending on heap memory allocations. This paper discusses experience with two experimental implementations of a safety critical application using the PERC Pico technology.

Keywords: language, ARINC 653, Java, safety, certification

1. Introduction

DIANA [1], Distributed, equipment Independent environment for Advanced avioNics Applications, is an aeronautical research and development project funded through the European Commission's 6th Framework Programme and led by Skysoft, Portugal. It aims at the definition of an advanced avionics platform, called AIDA (Architecture for Independent Distributed Avionics), supporting execution of object-oriented applications over virtual machines, secure distribution services, and a tool chain supporting model-driven engineering.
The overall objective of AIDA is to reduce the costs of avionics software development thanks to reduced development, validation, and integration efforts. Time reduction is achieved by using efficient infrastructure mechanisms like data distribution services inspired by the OMG standard DDS [2]; modern development approaches like the use of models inspired by the OMG standard Model Driven Architecture (MDA) [3] and programming languages in combination with an architecturally Neutral Execution Platform (NEP) to enable efficient software reuse and integration of independently developed software components [6]. This paper focuses on the benefits of the NEP.
One of DIANA's challenges is the deployment of Java applications on the AIDA platform. To address this challenge, AIDA relies on the PERC Pico technology developed by Atego. PERC Pico is an original solution addressing concerns raised previously [] regarding the memory management approaches JSR 302 [5]. Note that JSR 302 aims to define a Java specification for safety critical development. The Atego PERC Pico solution is original in the sense that differs slightly from the JSR 302 approach with respect to the syntaxes used by programmers to manage dynamic memory allocation and deallocation. The architecture of the AIDA NEP, based on PERC Pico, and its tool chain is an important topic of this paper.
In the scope of the DIANA project, Atego has ported PERC Pico to the ARINC 653 Application Executive [4], the standard real-time operating system specification in the avionics domain, so as to ensure backward compatibility for ARINC 653 compliant applications. Challenges encountered during this porting effort concern differences between Java and APEX synchronization, priority inversion, and error handling behaviors. How these challenges were addressed is another important topic of this paper. The NEP is validated by two demonstrators developed in the scope of the DIANA project:
a cabin air-conditioning application developed by the National Aerospace Laboratory (NLR)
a simplified Flight Warning System (FWS) developed by THALES.

2. AIDA Neutral Execution Platform Concepts

2.1 Objectives

The general idea is to set up a process where software development would start using full-fledged Java (including language constructs, run-time environment, Application Programming Interfaces) and would move progressively to a real-time version of Java. The goals are to (i) benefit from the many features of standard Java to achieve high levels of generality and shortened development cycles during the initial development phases, and (ii) to benefit from the determinism and safety of real-time

versions of Java during the later development phases.
This approach is commonly known as separation of concerns: we would like the developer to focus on functional requirements first, without the burden of addressing timeliness or memory usage concerns, and consider the real-time resource utilization aspects later, after the functional software specification and low level design are “frozen”.

2.2 AIDA concepts

AIDA is based on the concepts of Integrated Modular Avionics (IMA). Time and space partitioning is therefore one of its main architectural characteristics. This partitioning separates applications from each other by means of memory protection, and time slicing according to a fixed cyclic schedule.
The underlying RTOS is expected to enforce time and space partitioning and to provide the ARINC 653 API (known as the APplication EXecutive or APEX) as well as health monitoring of hosted applications. No other assumptions are made about the operating system (OS).
AIDA is an interoperability platform. It provides support for data exchange and service interoperability. The objective is to ease the integration of distributed applications. Each partition provides or requests data through well-defined interfaces and is integrated with other components by means of configuration files.
In the traditional data processing industry, reusability is usually understood as code reusability. To achieve real commercial advantages in the context of safety-critical systems, reusability should also cover reusability of certification credits [7]. Whereas certification reuse for software entities like operating systems is reality today, there are still many obstacles for applications. One of the problems is that an application is not guaranteed to produce the same behavior when executed on different implementations of an execution environment (like an ARINC 653 OS). Because of subtle differences between different implementations of the ARINC 653 standard, for example, certification evidence collected for a given safety critical application running on one implementation of ARINC 653 provides no assurances that the same application will run in a safe manner on a different implementation of ARINC 653.

It does not appear realistic to achieve strict neutrality in the near future, and we do not claim to have found a complete solution to this problem. Instead, this research investigates proposed means to raise the level of neutrality. Specifically, the goal is to reduce the amount of certification efforts (such as reworking of specification, design, code, verification, and validation) associated with porting an application from one environment to another.
The use of a Neutral Execution Platform (NEP) to decouple the application from the underlying execution environment is the key mechanism studied in this research., This NEP assures that for any given set of inputs, the application will produce the same outputs on every underlying operating system. A limitation of neutrality is that the NEP does not necessarily assure that all platforms will produce outputs at exactly the same times.

.
The NEP is abstractly defined in terms of computational states on program level (Initialization, Mission, Recovery) and thread level (Dormant, Waiting/Blocked, Ready, Running), a scheduling policy, and standardized services that provide concurrency, memory and time management. This abstract specification resembles other specifications like the ARINC 653 APEX. However, the NEP is meant to avoid platform dependent behavior, introduced by differences between implementations or special properties of low-level programming languages such as the C language.
The DIANA project exploits the results of other European projects, like HIDOORS and HIJA [8], proposing Java virtual machines to be used for the NEP layer. In the scope of the project, Atego’s safety critical Java implementation PERC Pico has been selected to act as the AIDA NEP.

3. PERC Pico concepts

PERC Pico is a new product offering by Atego, designed to address the needs of very low-level and safety-critical development with the Java language. Structured as a subset of the full Real-Time Specification for Java, PERC Pico approaches derive from early work within the Open Group to establish an open standard for safety-critical development with the Java language.
The initial drafts of the annotation and static analysis techniques and the API subset of PERC Pico were first developed and published by one of the authors (Kelvin Nilsen) in the first half of 2004, in response to a request from the Open Group’s Real-Time Forum subcommittee on safety-critical Java to design a “new memory construct, something beyond immortal memory, similar to scoped memory, but without run-time hazards.” In subsequent meetings that same year, the approaches were refined in response to community feedback.
A commercial implementation of the proposed mechanisms has been available since March 2007 in the form of the PERC Pico product by Atego. This technology was released for public use in order to encourage experimentation and gather industry feedback in order to refine the technology and influence the emerging standards.

To date, the PERC Pico product has not yet been deployed in any commercial products. Neither has the compiler been qualified nor the run-time environment certified to established safety-critical guidelines. Currently, the product is used primarily for experimentation and research in several domains, including applications in rail transport, avionics, defence systems, and orbiting satellites. Atego highly values the feedback received during these experimental studies. As a result of interactions with early evaluators, Atego has made several refinements to the product, adding support for dynamic class loading, adding libraries for collections, and refining the specification in order to broaden the set of Java programs that are considered legal according to the rules of the PERC Pico byte-code verifier.

The early safety-critical Java standardization work carried out by the Open Group has been handed over to the Java Community Process under JSR-302, with the Open Group assuming the role of specification lead. Work on the JSR-302 standard is ongoing. In order to establish consensus on the draft standard, the JSR-302 expert group has chosen not to seek standardization of certain techniques for development of safety-critical Java code. The annotation and byte-code verification system originally designed and refined within the Open Group process is no longer part of the draft JSR-302 standard. Rather, the draft specification acknowledges that many issues that must be addressed by a safety-critical developer are not addressed in full by the JSR-302 specification. The draft JSR-302 standard suggests that developers work with development tool and virtual machine vendors to address these issues using vendor-specific (non-standardized) solutions. The annotation and verification system provided by PERC Pico is an example of such a vendor-specific solution.

The key issues addressed by the PERC Pico annotation system are as follows:

Assure that scope memory relationship constraints associated with all arguments passed in to a method are clearly identified so that all users of the method understand the constraints required for reliable execution of the method.

Assure that all arguments passed to a method satisfy the scope memory relationship constraints associated with the invoked method.

Reduce the syntactic clutter associated with entering and exiting scopes and establishing scope sizes, with the objective of making code more readable and more maintainable and reducing the opportunity for human error.

Enable modular composition and maintainability of real-time software components by establishing clear separation of concerns between independently developed software modules, by inhibiting incompatible evolution of independently maintained software modules, and by automating the budgeting of resources (such as memory and CPU time) that may be shared between multiple independently maintained modules.

PERC Pico builds on the notion of the Real-Time Specification for Java's (RTSJ) ScopedMemory. However, allocating, entering, and exiting scopes is implicit, controlled by syntactic annotations instead of explicit API calls. Conceptually, every method introduces a new private scope. However, no scope is created for methods that perform no local allocations. Besides simplifying the syntax, the main benefit of this approach is that it enables unambiguous static analysis of code to determine which scopes are relevant to each memory allocation request. If the RTSJ ScopedMemory APIs were exposed to the developer, the static analyzer would need to determine for each new memory allocation request the API invocation history that precedes the allocation in order to determine which scope is to supply memory for the allocation. Theoretically, this reduces to the halting problem, which is generally understood to be unsolvable by a static analysis tool.

PERC Pico uses Java 1.5 meta-data annotations to augment the traditional Java type system with information describing the scoped memory constraints associated with method arguments and results. Consider the following class definition as an example:

```
public class Complex {
        float r, i;
        @ScopedThis
        public Complex(float r, float i) {
                real = r; imaginary = i;
        }
        @CallerAllocatedResult @ScopedPure
        public multiply(Complex arg) {
                float r, i;
                r = this.r*arg.r – this.i*arg.i;
                i = this.r*arg.i + this.i*arg.r;
                return new Complex(r, i);
        }
}
```

The Complex constructor is declared @ScopedThis, indicating the programmer's intention to allow this constructor to apply to Complex objects allocated within ScopedMemory. The PERC Pico verifier enforces that the body of this constructor does nothing with its implicit this argument that would violate the expectation that the constructed object may have a temporary lifetime. In particular, the verifiers forbids the constructor from saving the value of this in any static variables or in any instance fields associated with objects that cannot be proven to

have a shorter lifetime than the constructed object itself. The multiply() method is declared to treat both this and arg as possibly residing in scoped memory. Furthermore, the @CallerAllocatedResult annotation denotes that the result returned from multiply() is allocated in a scope external to the method itself.
Code written to satisfy the PERC Pico verifier will also run on a standard edition Java virtual machine, using heap allocation and garbage collection to manage temporary objects, provided that all of the referenced libraries are available. Based on programmer-supplied annotations, the PERC Pico verifier determines whether there is a compatible sizing and use of memory scopes that would enable the same code to run reliably in an environment that uses ScopedMemory rather than garbage collection to satisfy temporary memory allocation needs. In performing this analysis, the PERC Pico verifier automatically determines for each memory allocation which scope to take the memory from. Its preference is generally to satisfy every memory allocation from the most local scope that is relevant to the request.
Assume existence of a static method declared as shown below:
void doComplex(Complex arg);
Note that the argument is not declared to be @Scoped. Suppose the PERC Pico verifier is tasked with analyzing the following code sequence:

```
Complex a, b, i;

a = new Complex(3.5, 2.4);
i = new Complex(0, 1);
b = i.multiply(i);
doComplex(a);
```

Note that the above code allocates three Complex objects which it assigns respectively to the a, i, and b variables. The PERC Pico verifier's goal in this case is to prove that it is safe to perform all allocations within this method's local scope. The verifier identifies a problem when it observes that the temporary object assigned to local variable a is passed as an argument to the doComplex() method. Since that method does not declare its argument to be @Scoped, the verifier concludes that in this context, it is not safe to allocate object a in ScopedMemory. In response to the error message produced by the PERC Pico verifier, the programmer has two options. Either he can annotate this context to allow the object a to be allocated in ImmortalMemory, or he can add the @Scoped annotation to the argument of doComplex() and then deal with any additional issues that might surface within the implementation of doComplex() as a result of making that change.

4. The Avionics demonstrator

4.1 Objectives

Within the DIANA project, two demonstrators have been developed to exercise and validate the project's specific mechanisms including the NEP, the configuration selection mechanisms, the communication middleware, and others. In this context, the demonstrator developed by THALES DAv (Avionics Division) is strongly focused on the use and benefits of Java as the backbone of the NEP.
THALES interest on Java is not new. In particular, the work presented in this paper is the continuation of previous achievements in the HIJA European project [9]. Our main objective is actually to (i) contribute to the acceptance of Java as a language for embedded real-time systems, and (ii) validate the feasibility of a "smooth" and iterative development process based on Java.
Concerning the latter point, we experimented with a process where software development starts using full-fledged Java (including language constructs, full run-time, and the complete standard edition Java libraries) and moves *incrementally* to a constrained version of Java. In this experiment, constraints are essentially targeted towards a better determinism of memory management, in time and space. This way, we expect (i) to benefit from the many features of standard Java to achieve a high level of generality and ease of development during the initial project phases, and (ii) to benefit from the determinism and safety of real-time versions of Java during the later development phases.
For instance, we would like the developer to focus on the functional requirements first, without the distraction of worrying about timeliness or memory usage. Once the software specification and low-level design are frozen, developers turn their attention to the real-time constraints on time and memory resources. This approach represents an application of the general software engineering principle known as separation of concerns.
As an introduction to this topic, several questions are addressed below:

- There is a general tendency to organize software development around models and their cohorts of semi-formal languages, one clear objective being to facilitate the automatic generation of code. In that context, is the choice of a programming language still an important issue?
  Indeed, automatic code generation is effectively a very efficient means to shorten the "specification to product" path. However, this is only a practical solution for few parts of most applications (e.g., framework-generated code), or most parts of few applications (e.g., SCADE-

generated code). In particular, it does not seem to be a solution for applications involving complex data structures and data processing. As of today, and for the next few years, manual coding is still the only solution for most developments.

- *Assuming that the programming language is actually an issue, why use Java instead of Ada 95, Ada 2005, or C?.* Even though such question is legitimate, we will not discuss it. The choice of a programming language is subjective. This topic is discussed at length in other forums. In the context of our experiment, we simply considered the opinion of the people in charge of prototyping new functions in our service: they selected Java for *some* reasons, and we consider those reasons to be *good*.

- The proposed approach seems to rely on the existence of a continuous transformation process starting from some "quick-and-dirty" code and ending, magically, with some high quality code. Isn't it contradictory with all usual and admitted practices?
  In fact, we are certainly not promoting such a practice. In our context, "transformation" means the "progressive" introduction of concerns at the most appropriate phase of the development process. Stated another way, we simply want efforts to be focused on the most important and unstable "functional" aspects first, and then on the other, "non-functional" transverse aspects. Naturally, this operation will require refactoring, recoding, retesting etc. but we expect the benefits to be worth the additional efforts.

4.2 The target application

For the demonstration purposes, we selected the core of the Flight Warning System (FWS) as the target application. Briefly, the FWS performs the following main functions:
It acquires state data from other aircraft systems and determines the occurrence of an abnormal or emergency situation
It elaborates and manages warning and cautions alert messages that are displayed on the Engine and Warning Display (EWD)
It manages the interaction with the crew, via attention-getter means (aural, master warning light, master caution light) and a set of buttons located on the Electronic Flight Control Panel (EFCP). The first phase is data input and simple computations. In the actual product, this code is generated automatically from a semi-formal SCADE model and using Java would not bring any significant benefits. So, the experiment focused the second and third phases.
During our experiment, focus has been placed on memory management. In particular, many important aspects such as threading, inter-process communication, or testability have not been addressed.
The FWS shows fairly simple data flow and object creation / destruction profile: object creations occur essentially during an initialization phase that sets up all the internal data structures representing the alerts, their relationships (priority), their associated messages, their aspect (e.g., color).
It also shows a very simple threading model; the application uses a single periodic thread that acquires the equipment state (including buttons), determines the new alerts, manages the actions of the crew and generates messages to be displayed.
Consequently, extrapolations of evaluation results to other applications, especially more complex ones, must be done with extreme care.
The FWS development was a very good illustration of our approach. Indeed, a first version of the application written in standard Java was already available. It had been developed to confirm customers' requirements, and validate particular design choices.
The experiment carried out in DIANA essentially focus on memory management aspects, for the following reasons:
Memory management is a complex and error prone task. Any feature that simplifies this task is considered to have a strong impact on productivity and quality.
Memory management is often mentioned as one of the most important reasons not to use Java in embedded systems.
Memory management is the domain where Pico is the most innovative. Note that the applicability and the application cost of recommended practices for the usage of object oriented language in avionics — such as those defined in the OOTiA guidelines [10] — have not been considered yet. We consider that, for the most part, those constraints do not depend on the programming language and, consequently, are not impacted by the selection of Java instead of Ada95, for instance.

4.3 How we used PERC Pico

Our activity has essentially been a porting activity, i.e., the adaptation of an existing code to make it comply with some real-time constraints. Some specific development was required when the platform mechanisms (e.g., serialization) or API (e.g., containers) used by the original software were not provided by PERC Pico or were not stable enough. In that case, development was kept as simple as possible.
During this porting process, we imposed on ourselves a constraint to minimize the modifications to the original Java application, so as to be as representative as possible of the way we would actually use Pico in our anticipated operational

context. This is a very important point, since it means that we have had to solve issues rather than simply preventing them.
Thanks to the simplicity of the application, our difficulties have been concentrated on two points: to ensure referential integrity, and to estimate memory usage.

4.3.1 Ensuring referential integrity

Referential integrity is guaranteed when the application passes all verifications applied by Pico's verifier tool. To achieve this objective, we adopted a very pragmatic, iterative, approach relying on the information provided by the PERC Pico verifier and on our knowledge of the application:

- Use previous verification results (if any) and operational knowledge of the application to initiate or refine the application's annotations
- Check annotations using Pico's verifier
- If all checks pass or if no error diagnosis has disappeared since the previous verification, go to next phase, otherwise, loop to phase 1
- Analyze all remaining irreducible error diagnosis and demonstrate that they correspond to conditions that can *never* occur in operation. By "demonstration", we mean the provision of a sufficient set of convincing elements. In the case that developers are able to assert that the PERC Pico verifier is overly concerned about certain issues, the developers insert additional annotations to advise the PERC Pico verifier to relax its normal enforcement in these special circumstances.

To initiate the process (first phase 1), we perform a first and rough annotation phase where:

- We first identify all objects whose life spans the lifetime of the application, and we place them in immortal memory. For the remaining objects, we analyze the benefit/cost of putting them in immortal memory, both from a memory sizing point of view and from the impact of this choice to the allocation site of potentially related objects.
- Once the previous phase is complete, only scoped objects remain. So we identify *captive* references to scoped objects. A captive reference is one that is known not to escape the context in which it was allocated. To do so, we use both a manual (and painful) analysis of the code's data flow, and the verifier itself. In the latter case, we put the strongest constraint ("captive scoped") on arguments and relax those constraints (to "scope") until the analyzer does not reveal any more error.

If this solution is "pragmatic", it is costly, and clearly not acceptable for a new development. In our specific case, this was really helpful since the relations between objects *in the existing application* were not fully formalized and described. Important methodological questions about the way lifespans of objects must be considered during design and coding would deserve further studies. Provision of *design patterns* would certainly help.

4.3.2 Estimating memory usage

As of today, PERC Pico does not provide an automatic and static means to estimate memory, though the capability has been designed and is anticipated in a future product release. Instead, it currently relies on software developers to insert scope sizing annotations or invocations of memory estimation methods ("sizer()") and objects ("SizeEstimators"). These elements are used to modularize memory estimation, and provide an upper-bound of memory usage taking into account all state variables that may have an impact on memory usage, including the sizes of all classes that might need to be instantiated within this scope. In practice, we first estimate a lower bound of memory usage by observing the worst case heap usage on a standard virtual machine. Once this information is obtained, each class is adorned with an appropriate "sizer" method which may refer to other classes' sizers, recursively.
A difficulty here lies in the fact that to estimate the size of memory areas, one has to know in which of these memory areas PERC Pico decides to place the objects.

4.4 Benefits and issues

4.4.1 Benefits

Transition from standard Java to PERC Pico requires very few modifications of the original software architecture and code. In particular, the application remains mainly free from memory management code (creation, deletion). Even though "sizer()" methods" or "SizeEstimators" objects are needed to support memory usage estimation, they interfere only lightly with the application code. This has to be compared to the RTSJ where memory management is essentially under the developer's control: he / she has to create the scoped memory areas, to create the Runnable covering the code to be executed in the memory area, to enter the scope, to take care of the reference transfers between memory areas, etc.
Capability to run PERC Pico annotated Java code on a standard Java virtual machine via some simple API stubbing is another interesting property. It makes it very easy to execute regression tests using all the

capabilities of a standard Java platform. It also allows a very easy switching between the standard Java and PERC Pico development phases. Furthermore, it ensures the ported code to be usable in *any* Java platform, including those that rely on a real-time garbage collector.

PERC Pico supports a modular management of memory. Modularity is supported by the annotation scheme which is used to express a memory management contract between a method *user* and a method *provider*. This represents a significant advantage to the RTSJ's scopes, which introduce strong and implicit dependencies between different code parts, and makes the application very fragile.

The PERC Pico verifier guarantees that no referential integrity problem will ever occur once the verification is successful. This represents another significant advantage with respect to a RTSJ platform which cannot ensure the absence of referential integrity errors at compile time. Instead, it relies on runtime checks and appropriate exception handling by the user. Even though handling the exception allows a safe and smooth handling of errors, it does not ensure *per se* the correct behavior of the software. We have then to compare the extra development cost (annotations) of PERC Pico to the extra design and verification cost of the RTSJ, taking into account the fact that in the latter case, there will be no guarantee for safety.

4.4.2 Issues

The PERC Pico learning curve is steep, for various reasons, among which:

- The concepts, Pico is based on, are complex. They must be supported by a clear and rigorous documentation written for the end-user, that is to say the application developer.

- Translating those concepts into code is another difficulty that could well be alleviated thanks to design, coding patterns, or tools.

- Errors signaled by PERC Pico are hard to diagnose and correct. More often than not, one has to "play the role of the verifier" to understand why a particular error message is reported.

The latter issue is even more critical when considering the apparent *pessimism* of the analysis which translates to the following frequent observation: *The verifier rejects this assignment even though I **know** that the assignment will always comply with the RTSJ referential integrity condition.* In fact, the analyzer can only deduce potential errors from the information it has access to, i.e., the annotations and the code, just as any compiler would. So, the more expressive the annotations are the more accurate is the diagnosis. The lack of expressiveness of annotations, grounded to some extent to the limits of Java annotation mechanism, eventually leads the developer to consider the diagnosis as a "false positive" (to use a terminology borrowed from the abstract interpretation domain, in a way slightly abusive)[2], whereas it is simply a direct consequence of the interpretation of the annotations.

Additionally, if the approach based on "local" annotations expressing "local" properties is imposed by the requirement for modularity, it also leads to conservative estimation if those properties are insufficiently accurate. In practice, this means that the verifier may reject a statement because it *believes* (from what it "knows") that referential integrity cannot be ensured whereas the developer *knows* it for sure. Intuitively, the remedy seems to depend on an improvement the code annotation scheme, to allow more information to be provided by the developer to the analyzer, in the same way as a programming language with a sophisticated typing system will support a much better diagnosis than a language loosely typed, considering the same target level of safety at the end.

A "pessimistic diagnosis" is acceptable as long as (i) it does not occur too often and (ii) the difficulty to demonstrate that no failure can ever occur in operation despite the diagnosis is reasonable. Unfortunately, as of today, whenever the analyzer rejects a piece of code, a great deal of energy must be spent to understand the problem, to demonstrate its innocuousness or to refactor the code or the annotations to make it disappear. One clearly needs indication and hints to track the "origin" of the diagnosis.

Generally speaking, the other, and certainly better, solution would be for the designer to carefully take into account memory management constraints during the design, so as to ease the task of the "verifier" later[3]. However, this was not the context of our experiment, which takes as a working hypothesis to minimize the refactoring of the code.

The estimation of memory usage is another point where things could be improved. Currently, PERC Pico's documented annotations and mechanisms to

[2] Developers familiar with traditional static analysis or abstract interpretation tools tend to view PERC Pico's conservative enforcement of constraints as a misunderstanding of their intent. PERC Pico's support team tends to view this situation as an inconsistency between intent and annotation, requiring that programmers enhance or refine the annotations. This conflict of perspective highlights the subtle complexity of the verification system.

[3] In fact, the issue is more fundamental: PERC Pico relies on a computational model (a stack automaton) that is not as expressive as the one of standard Java (a Turing machine). In practice, this means that the "translation" of code may simply be impossible without a strong refactoring.

support static analysis of memory usage seem to be very promising but, unfortunately, they are not fully implemented. So, the developer ends up with code for which referential integrity is verified, but for which some memory allocation constraints may be violated, leading to, potential, very bad effects at the application level. So, and until Pico's memory estimation mechanisms are fully implemented, memory dimensioning still requires a "classical" verification approach (testing, inspection, etc.) which is certainly simpler than ensuring referential integrity, but which deserves a specific testing strategy. Furthermore, if memory usage estimation is "relatively" easy when the location of allocations is managed by the user, it is much harder when these locations are determined by the compilation chain.

Besides time and memory determinism, which have been addressed previously, certification, more generally, concerns compliance of Pico's runtime and libraries with the DO178 (currently, issue B and soon, issue C) objectives. This current version of PERC Pico does not come with any *qualification case*. However, even if we cannot show the artifacts required for certification (SDP, PSAC, SAS, etc.), data provided by Atego give hints on the fact that they *could* be obtained, and at a *reasonable cost*:

The runtime represents around 6700 Lines of Code (LOC).

For the C part, the runtime represents around 2700 LOC (2000 LOC for "port.c", which is customized for each operating system and 700 for platform.c).

The API represents around 27000 LOC for java.lang package, and 13000 LOC for the sc package.

These values are *reasonable* small with respect to the size of the applications themselves.

A more controversial point concerns the qualification level of the code generator and verifier. In particular, to what extent can we take credit from the verifications performed by the verifier to reduce the test activities on the generated code?

This questions leads to two observations. First, the complexity of the properties to be verified makes it very hard to trust the completeness of Pico's verifications, even though those properties are expressed at length in Pico's documentation. As the verifier not only checks compliance with referential integrity, but also manages the allocations, this process shall come with strong evidences that it is built on a sound —possibly, mathematical — basis. Second, verification and code generation processes seem to be tightly coupled: the code generator uses information provided by the code verifier to determine the scope. So, if there is an error in the verifier, (i) an error in the source code may be left undetected (the verifier is considered as a verification tool), (ii) the generated code itself may be erroneous (the verifier is considered as a part of the code generation process). It would be highly desirable to have a tighter integration between the PERC Pico verifier and the PERC Pico code generator, and to "qualify" this integration for use by safety critical developers.

More fundamentally, it seems that one can hardly consider Pico's verifier to be a "classical" *verification tool* according to the DO178 definition, so submitted to the significantly relaxed verification constraints. Indeed, the probability to introduce an error at memory management level (e.g., a reference assignment that violates referential integrity or a memory allocation that is lower than the actual memory usage) is (i) very high, and (ii) the error detection coverage of such errors by testing is fairly low. Consequently, even though a double error is needed for an error to show in the product, the probability of such event cannot be considered negligible.

Finally, since PERC Pico's static verifier does not operate on the final object code but on the intermediate Java byte code, verification activities targeted to memory management errors and applied on the final object code are still necessary. A testing strategy focused on memory management needs to be defined.

We end this brief analysis of PERC Pico's pros and cons by a last point about standardization. In effect, PERC Pico does not comply with the RTSJ or the future SCSJ standard (which is based on the RTSJ). From the strict perspective of *porting* standard Java applications, this should not be so strong an issue since the standard Java version of an application and the PERC Pico version only differ at annotation level. In this regard, moving from PERC Pico to the SCSJ (JSR-302) will be no more complicated than moving from standard Java to the RTSJ. However, this divergence is actually a problem in the general case. Convergence of PERC Pico APIs to the SCSJ, and the capability to make both SCSJ and PERC Pico applications to cohabit in different SCSJ missions would offer an opportunity to benefit from both technologies. Atego has announced that this is their intended evolution of PERC Pico once the JSR-302 standard is finalized.

5. Conclusion

In this paper, we have briefly presented the main features, some specific porting aspects and some evaluation results of PERC Pico, Atego' implementation of Java for real-time safety critical systems.

Java and its implementation in PERC Pico is a key technological brick of DIANA's neutral execution platform. It decouples the application code from the execution platform in a more complete way than any application / executive API would. The development of the JVM to APEX interface raises few difficulties concentrated on very few points, such as threading and priority inheritance in particular. This work being done, PERC Pico can now be used on any APEX compliant platform.

Concerning the application to JVM interface, PERC Pico's modular approach of memory management supports a smooth transition from a code using standard Java memory management model to a code compatible with the safety and determinism constraints of real-time avionics systems. This was an important objective of the experiment.

PERC Pico relies on a simple memory model, a set of annotations, and a powerful static verification tool. It is expected to compare advantageously to solutions based on scoped memory, in particular when modularity and runtime safety — or, the other way round, testing effort — are concerned.

However, an important effort is necessary to become familiar with the PERC Pico concepts, and become proficient in their application. For the most part, this is due to the difficulties to understand the verifiers' diagnostics, and to translate them to corrections at the application level. This leads to the paradoxical situation where memory management issues are first removed from the developer considerations, but reintroduced in a later development phase. In this later development phase, it becomes necessary to insert scope annotations, dimension the sizes of memory areas, and deal with any "false positive" error messages reported by the PERC Pico verifier. The challenges of this later development phase significantly reduce the potential benefits of the approach.

Concerning certification, although this aspect has not been completely covered yet, the simplicity of PERC Pico should strongly limit the effort to establish a qualification case. However, some interrogations remain concerning the completeness of the verifications performed by PERC Pico verifier, the relation between code production and verification, and the actual qualification credit that can be sought using this tool.

A programming language is a long-term investment. To introduce a new language, many aspects shall be taken into account: technical aspects such as expressiveness, performance, robustness, safety, etc., but also non technical aspects such as availability of educated developers, availability and maintainability of development environments, tools, libraries, etc. Regarding many of those aspects, Java is a very good candidate, and some important and recurrent criticisms of Java are now obsolete thanks to the combined efforts on virtual machines, verification means, and runtime libraries. What is still missing for a wider acceptance of Java in the mission critical domain is a common and collaborative effort of industrial partners and technology providers towards a runtime and a set of API components developed according to applicable development standards (e.g., DO178C, CEI 61508 and derivatives).

6. Acknowledgement

We thank Olivier Charrier, from Wind River who helped us running the avionics demonstrator on VxWorks653.

7. References

[1] DIANA IST- FP6 project, http://diana.skysoft.pt

[2] Object Management Group, Data Distribution Service for Real-time Systems, OMG specification, omg/07-01-01.

[3] Object Management Group, Model Driven Architecture guide, omg/03-06-01.

[4] Airlines Electronic Engineering Committee (AEEC), Avionics Application Software Standard Interface (ARINC Specification 653 Part 1 – Required Services), ARINC Inc., March 2006.

[5] JSR 302 – http://jcp.org/en/jsr/detail?id=302

[6] Szyperski, C, Component Software: Beyond Object-Oriented Programming. 2nd ed. Addison-Wesley Professional, Boston, 2002.

[7] Federal Aviation Administration, Advisory Circular on Reusable Software Components (= AC 20-RSC), June, 2003.

[8] The HIJA consortium: High Integrity Java, project home page at: http://www.hija.info

[9] Hu, E. Y., Jenn, E., Valot, N., and Alonso, A. 2006. Safety critical applications and hard real-time profile for Java: a case study in avionics. In Proceedings of JTRES '06, vol. 177. ACM, New York, NY, 125-134.

[10] Federal Aviation Administration, Object Oriented Technology in Aviation (OOTiA), Vol. 1-4, http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot/.

8. Glossary

AIDA:	Architecture for Independent Distributed Avionics
APEX:	ARINC 653 Application Executive
ARINC 653:	Avionics Application Software Standard Interface
FWS:	Flight Warning System
JVM:	Java Virtual Machine
JSR 302:	Java Specification Request in charge of defining the Safety Critical Specification for Java
NEP:	Neutral Execution Platform
PERC Pico:	Virtual Machine for Java real-time and safety critical systems
RTSJ:	Real Time Specification for Java
SCSJ:	Safety Critical Specification for Java