

# Re-engineering a safety-critical application using SPARK 95 and GNORT

Roderick Chapman<sup>1</sup> and Robert Dewar<sup>2</sup>

<sup>1</sup> Praxis Critical Systems Limited, 20 Manvers St., Bath BA1 1PX, UK,  
rod.chapman@praxis-cs.co.uk,  
<http://www.praxis-cs.co.uk/>

<sup>2</sup> Ada Core Technologies Inc., 73 Fifth Avenue, 11B, New York, NY 10003, USA,  
dewar@gnat.com,  
<http://www.gnat.com/>

**Abstract.** This paper describes a new development of the GNAT Ada95 compilation system (GNORT) that is appropriate for the development of high integrity embedded systems. We describe GNORT, the motivation for its development, and give some technical detail of its implementation. The latter part of the paper goes on to describe SHOLIS—an existing safety-critical application written in SPARK 83 that has been re-engineered to take advantage of SPARK 95 and GNORT. We assess the benefits of this approach through metrics on the SHOLIS application source and object code. These data may be of interest to engineers who are considering Ada95 for a new project or converting an existing Ada83 application to Ada95.

**Keywords.** High Integrity Systems. Ada Language and Tools.

Reproduced from Reliable Software Technologies Ada Europe '99. Lecture Notes in Computer Science, Volume 1622, June 1999. pp. 39 - 51. (c) Springer-Verlag.

## 1 Introduction

This paper describes GNAT-No-Runtime (GNORT)—a development of the GNAT Ada 95 compiler that is appropriate for the production of high integrity embedded systems. We also describe an existing safety-critical application (SHOLIS) which has been re-engineered using GNORT. This work was carried out with two major goals:

- To evaluate if the subset of Ada 95 allowed by GNORT is a superset of the language supported by SPARK 95,<sup>1</sup> and to provide its developers with early feedback on its performance when used to compile a real-world application.
- To evaluate the improvement (in terms of program size, object code quality etc.) that can be gained from re-engineering an existing application using SPARK 95 and GNORT.

---

<sup>1</sup> The SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on the SPARC™ architecture.

Implementations of Ada95 are now beginning to appear which are (at least claimed to be) suitable for high integrity systems. Engineers may be considering Ada95 for new projects, or converting existing applications to Ada95, but there is little empirical evidence to suggest what benefits may be gained from such a change. This paper hopes to address this need, at least within the context of high integrity, embedded systems.

The remainder of the paper is structured as follows. Section 2 describes GNORT, covering its goals, design and implementation, and gives a comparison with other high integrity Ada compilation systems. Section 3 gives a brief outline of the SHOLIS system. Section 4 describes how SHOLIS was re-engineered to take advantage of SPARK 95 and the facilities offered by GNORT. Sections 5 and 6 offer thoughts on further work and conclusions.

## **2 GNORT**

### **2.1 The motivation for the development of GNORT**

The notion of a “run time library” is a relatively new one that has appeared in connection with more advanced high level languages. Certainly there were library routines accessed by a typical Fortran program, but the development of complex language semantics (for example tasking and exception handling) that require run-time code for support of basic language semantics, introduces new considerations into the development of safety-critical code that must be certified.

Two approaches are possible. Either the run-time library must itself be certified, or it must be eliminated. Some other Ada systems have pursued the first approach, but not only is this an expensive process, possibly requiring major reengineering of the run-time, but it is also procedurally awkward, because it means that the ultimate application must depend on a “foreign” certification, and in particular, it is difficult to deal with varying certification requirements.

For these reasons, we decided to adopt the second approach, of eliminating the run-time system entirely. This completely solves the certification problem, since there is no run-time to be certified, and the entire code of the executable image is derived directly from the customer’s application, and certified in a manner that meets the particular certification needs of the application.

The only disadvantage of this approach is that it requires the language to be fairly severely subsetted. Front-end features like generics, child packages, derived types etc are not affected, but fundamental run-time facilities such as tasking and exception handling must be eliminated. However, in practice many, if not most, safety-critical applications are already committed to using a small subset of the language for certification reasons, so in practice the limitations introduced by this subsetting may be entirely acceptable. As a measure of this, it is interesting to note that GNORT can handle all the Ada language constructs that appear in the SPARK language[5].

### **2.2 Potential uses of GNORT in the development of high integrity systems**

The use of Ada in high integrity systems is attractive for many reasons. In particular the strong abstraction facilities, and strong typing semantics of Ada, which mean that many

errors can be caught at compile time, can substantially contribute to the reliability of applications, and thus the ease of generating high integrity systems. It is important to note that we are concentrating here on the static semantic facilities of Ada that provide compile time security, not on the more elaborate run-time facilities, such as exceptions, which are unlikely to be used anyway in safety-critical systems.

Clearly it is feasible to develop high integrity systems using much lower level languages that completely lack a run-time system, such as C. The use of GNORT means that the run-time model is roughly comparable to that of C, that is it is constrained to be very simple. However, the great advantage of GNORT is that this simplicity can be achieved without sacrificing the careful high-level design of Ada 95 that provides excellent abstraction facilities and strong compile time checking capabilities.

The fact that GNAT is committed to using standard system formats for all generated files, including the output of debugging information means that a very simple compilation model can be used with GNORT. The Ada units are simply compiled, and standard system format object files are produced. These object files can then be processed using any standard tool chain. The use of Ada in this manner does not introduce any Ada-specific difficulties in terms of the tool chain or run-time environment. The debugging information is carefully designed to be usable with a C-style debugger, but contains encodings that a more elaborate Ada knowledgeable debugger can use to provide full access to Ada data structures

### 2.3 The implementation of GNORT

The first step in the implementation of GNORT was to provide a configuration pragma *No\_Run\_Time* that restricts the features of the language that can be used. Basically this acts like a set of Restrictions pragmas, and causes fatal errors to be issued if any features are used that would require the run-time library to be accessed. The binder was also modified to check that this pragma is used uniformly on all units in a partition, to eliminate the automatic inclusion of a basic set of run-time routines, and to generate an entirely stand-alone main program.

Once these modifications were completed, a problem became immediately apparent, in that the subset of the language that could be supported depended on whether inlining was active. Inlining is activated in Ada 95 by the use of the *Inline* pragma, but in addition GNAT requires that the *-gnatn* switch be specified. In the presence of *-gnatn*, a number of important features, including dynamic dispatching, could be accommodated in GNORT mode, since the bodies of the run-time routines, consisting of a few machine instructions, could be inlined into the generated code, eliminating the need for the run-time routine itself.

To take advantage of this opportunity to extend the subset that could be accommodated, we implemented a pragma *Inline\_Always* that would inline even if *-gnatn* were not specified, and applied this pragma to a number of critical routines, thus significantly extending the subset that could be handled in GNORT mode. A corresponding change was required in the binder, to understand that even though a routine from a run-time library unit had been called, the run-time unit itself was not required after the inlining of the body.

## 2.4 GNORT and other high integrity Ada compilation systems

GNORT represents a simple and cost-effective approach to meeting the requirement for certified high- integrity code. The subset provided is large enough to allow most of the important facilities of Ada to be used. It is interesting to compare GNORT with the Aonix C-SMART system[4]. Both these systems provide a subset of Ada, and indeed the subsets are very similar. The important difference is that the Aonix approach requires a small run-time system, and certifying even a small run-time system is an expensive proposition, so this approach is definitely not a low-cost one.

Aonix has also recently announced the Raven product, which extends the system to include the Ravenscar profile[6] for limited tasking. This requires additional run-time support. There are two possible extensions to the GNAT technology to provide similar capabilities. The first, called GNARP (GNAT No-Runtime with Ravenscar Profile) relies on the use of the Java VM in the context of the JGNAT (Ada 95 to JVM) product. The native tasking provided by the JVM is very simple, but is powerful enough to support in-line generation of all the Ravenscar tasking constructs. In conjunction with a Java chip that provides this tasking support in hardware, this means that the pure GNORT approach can be extended to cover the Ravenscar Profile.

In other environments, a different approach is required. GNAT now provides a restricted run-time option that corresponds almost exactly to the Ravenscar profile. Our future development will simplify this restricted run-time, and we can either provide a certified component that implements this run-time, or, probably more practically, document the very simple interface required, and expect the application to provide the necessary simple tasking constructs.

One more comment that is relevant here is that the open-source model of software distribution embraced by GNAT is particularly well suited to our no run-time approach. In a proprietary model, the fact that GNORT is in a sense “nothing”, would make it difficult to charge money for the product itself. On the other hand, the open source model in which the charge is primarily for support, can accommodate this approach with no difficulty. Ada Core Technologies charges a modest (25% additional) fee over the normal support fee to provide full support for the GNORT feature, to cover the special additional support issues raised by the use of embedded tool chains.

## 3 SHOLIS

The Ship/Helicopter Operational Limits Instrumentation System (SHOLIS) is a ship-borne computer system that advises ship’s crew on the safety of helicopter operations under various scenarios. It is a fault-tolerant, real-time, embedded system and is the first system constructed to meet the requirements of Interim Defence Standard 00-55[1] for safety-critical software.

IDS 00-55 sets some bold challenges: it calls for formalised safety management and quality systems, formal specification of the systems behaviour, formal proof (at both the specification and code levels), fully independent verification and validation, and static analysis of program properties such as information flow, timing and memory usage. This section cannot hope to cover all aspects of the SHOLIS development, but instead concentrates on the construction of the SHOLIS software and its use of Ada.

### 3.1 The SHOLIS software

IDS 00-55 calls for the use of a small, rigorously defined programming language. To this end, SHOLIS is constructed almost entirely in SPARK 83[2]—a subset of Ada83 augmented with formal annotations which allow static analysis and program proof. SPARK eliminates language features which are not amenable to formal definition, program proof, and static analysis, or which might give rise to unpredictable behaviour in terms of run-time or memory usage. SPARK also eliminates almost all of Ada83's implementation dependencies, so a SPARK program exhibits no dependence on parameter passing mechanism, expression evaluation order, or elaboration order. These restrictions are checked by the Examiner tool, which also performs data- and information-flow analysis, and generates verification conditions for program proof.

SHOLIS is not a trivial program. It comprises some 133000 lines of code, including some 13000 declarations and 14000 statements. The compiler used on the SHOLIS project was the Alsys (now Aonix) Ada83 to 68k cross compiler[3] with the "SMART" runtime system.[4] All optimisation options were disabled.

The principle challenges in the construction of the SHOLIS software were as follows:

*Real-Time vs. Provable code.* Writing software which is amenable to formal proof and which runs at an acceptable pace proved to be a major challenge. For example, a pure-functional programming style can be useful for proof purposes, but had to be rejected for SHOLIS, since the cost of returning large data structures from functions was deemed to be too expensive, both in terms of execution time and memory usage.

*Non-functional implementation dependencies.* SPARK eliminates most semantic implementation dependencies from Ada, but we found some non-functional properties which generated unacceptable code. For example, when initialising composite constants with a large aggregate, we found that the compiler generated elaboration code which allocated a temporary object on the heap, even if the aggregate could be evaluated at compile time. This was particularly unfortunate, since SMART does not feature a heap manager by default! These cases had to be "programmed around" by introducing field-by-field initialisation code for these objects, which would normally be considered poor programming style.

*Exception freedom.* Neither SPARK nor SMART feature exceptions, and the requirements for SHOLIS required static analysis to show that predefined exceptions could not be raised. The SPARK Examiner generates verification conditions, the proof of which ensure that a program is free from `Constraint_Error`, `Tasking_Error` and `Program_Error` are eliminated simply because language features that give rise to these are excluded from SPARK, or through simple static analyses performed by the Examiner. `Storage_Error` was eliminated through static analysis of the generated code to determine worst-case stack usage. As noted above, SHOLIS was carefully coded to avoid dynamic allocation of objects from a heap. These analyses also allow compiler-generated run-time checks to be justifiably switched off, which improves both the run-time performance of the system, and the developer's confidence in the system!

## 4 Re-engineering SHOLIS

SHOLIS is essentially constructed using what (to the Ada community) is considered to be “old” technology: SPARK 83 and an old (but well-respected and trusted) Ada83 compiler. We naturally ask “What benefit could be gained from re-working SHOLIS using SPARK 95 and a recent Ada95 compiler, such as GNORT?” The remainder of this paper hopes to address this question, using GNORT as a focus.

### 4.1 Phase 1 - Minimal port to SPARK 95

In phase 1 of this effort, a self-hosted version of GNORT was used to port the existing SHOLIS software to SPARK 95. Unsurprisingly, this required almost trivial effort: existing implementation-defined pragmas (e.g. *No Image*) were replaced with their Ada95 equivalents, and address representation clauses were adapted to be compatible with Ada95’s package System. Having completed these steps, GNORT compiled SHOLIS “first time.” GNORT did not indicate that any language features used in SHOLIS were incompatible with its supported language subset, which lent some evidence to our belief that the language subset compiled by GNORT is a superset of SPARK 95.

### 4.2 Phase 2 - Taking advantage of SPARK 95

Phase 1 resulted in a version of SHOLIS that was workable, but still offered scope for improvement through the adoption of new features of SPARK 95[5]. For example, SPARK 95 supports the “use type” clause, arbitrary deferred constants, and a partial data-flow analysis option which makes subprogram annotations far smaller and easier to maintain in parts of a system where full information flow analysis is not required or useful. The following subsections report our findings.

**Use Type.** In SPARK, the “use” clause is not allowed, so explicit operator renaming is normally used in SPARK 83 programs to make the infix operators of a type directly visible as needed. The introduction of the use type clause in SPARK 95 allows such renaming to be eliminated. In SHOLIS, this simplification reduced the size of the application by some 270 source lines.

**Atomic.** In SHOLIS, approximately 100 library level variables represent registers and buffers of specific I/O devices. Most of these devices only allow access to these registers using specific sizes and alignments, and all input devices and readable control registers must be considered Volatile (i.e. we consider that the value of such a register can be changed by the environment.) Secondly, some I/O registers required multiple reads or writes—care had to be taken to ensure that the intended number of accesses to each device was preserved in the generated code. In the original version of SHOLIS, we simply relied on having the compiler’s optimiser switched off and careful coding to ensure that a variable’s size, alignment and volatility were respected.

Ada95 provides pragmas *Atomic* and *Volatile* for these purposes. These use of these pragmas preserve the intended dynamic semantics, while (perhaps more importantly) also allowing optimisation to be enabled.

**Aggregates in initialisation.** In section 3.1, we mentioned that some composite objects in SHOLIS had to be initialised field-by-field, rather than using an aggregate, to avoid the implicit allocation of a large temporary object on the heap. GNORT always allocates such temporaries on the stack, so the original code could be restored. While this change seems trivial, it actually has a beneficial effect upon program proof. Consider initialising an array field-by-field:

```
--# pre True;
for I in A_Index loop
  A(I) := 0;
  --# assert (forall J : A_Index, A_Index'First <= J and
  --#                                               J <= I -> A(J) = 0);
end loop;
--# post A = A_Type'(others => 0);
```

In SPARK's model of program proof, this code fragment has 3 basic paths (from the precondition to the assertion, from the assertion to the assertion, and from the assertion to the postcondition) and thus the Examiner generates 3 verification conditions (VCs) which must be discharged to show this fragment partially correct with respect to its precondition and postcondition. The proof of these VCs is actually non-trivial, and cannot be completed by our automatic VC Simplifier, and so must be discharged by hand using our interactive Proof Checker.

The alternative code

```
--# pre True;
A := A_Type'(others => 0);
--# post A = A_Type'(others => 0);
```

is rather better in this respect. This fragment has a single basic path, and the resulting VC is trivially discharged by the simplifier.

**Elaboration Control.** Through a variety of rules, a SPARK program cannot exhibit any implementation dependence on elaboration order. Nevertheless, it has still been useful to experiment with Ada95's Preelaborate and Pure pragmas to determine their potential usefulness. The various packages in SHOLIS fall into roughly one of three classes:

- Those which declare types and simple operations on those types only. These packages are typically "stateless" in that they declare no variables.
- Those which implement device and I/O drivers. These typically declare objects which are mapped to physical I/O devices using address representation clauses.
- Main packages. These implement the core state and operations of the SHOLIS system.

We found that packages in the first class were readily made Pure. This has the added effect of informing the compiler that all functions in such packages are also Pure, and thus have no side-effects. Unfortunately, we found that neither Preelaborate nor Pure could be applied to most packages in the second class. This owes to the normal form of an address representation clause:

```
for A'Address use To_Address(...);
```

The `To_Address` function is not a static function as defined by the LRM, and so is not pre-elaborable, but remains the only way of turning an integer literal into a value of type `System.Address` since, following the LRM's advice, `System.Address` is private.

Another related problem was the elaboration of large composite constants. These are initialised by an aggregate, which is not a static expression in Ada95 terms, and so a compiler is not obliged to even attempt to evaluate this aggregate at compile time. The elaboration cost of such a declaration can be a major source of confusion amongst novice Ada programmers, since it may well be Pre-elaborable, but its cost at runtime may be significant (with some compilers), or zero (with others.) The placement of such a constant (e.g. at library-level or nested within a subprogram) can also have a significant effect on program performance. GNORT does well in this respect—it often goes further than the LRM's requirements, and evaluates such aggregates at compile time.

**ROM Data and Deferred constants.** SHOLIS includes a database of operational scenarios, which is physically located on a single card containing a bank of FLASH EEPROMs. The database can be changed, upgraded, or improved at any time *in situ* without any change to the main application software.

In Ada terms, the database is represented as a single large package, which exports a set of memory-mapped objects. In SPARK 83, the usual idiom of declaring variables with address representation clauses is used. For the analysis performed by the Examiner, though, it is useful to indicate that the objects in question are really constants, and not variables (they are in a ROM after all!), so we introduce a shadow package specification, containing a constant declaration which mimics each database variable declaration. This gives us two versions of the database package: one which is compiled, and one which is submitted to the SPARK Examiner.

This trick is important because constants in SPARK play no role in dataflow or information flow analysis, and thus never appear in SPARK's global or derives annotations. If this trick were not employed, then the annotations for each subprogram in SHOLIS would be significantly larger, but would carry no additional useful information.

While this approach is workable, the analysis time for the shadow package is significant (the Examiner spends a significant time analysing the initialising expressions which are then never needed!). The need to have two package specifications “in sync” also complicates configuration control.

SPARK 95 allows deferred constants of any type, which offers a more elegant solution:

```
ClearScreenC : constant GraphicTypes.GraphicString;
...
private
  for ClearScreenC'Address use
    System.Storage_Elements.To_Address(...);
  pragma Import(Ada, ClearScreenC);
```

This approach has the advantage that only a single package specification is required for both compilation and analysis. Analysis time is also improved by a factor of approxi-

mately 3 using this approach for SHOLIS’s database package, which features over 400 such declarations. Finally, we note that declaring these objects as constants rather than variables might allow an optimiser to improve code generation for some algorithms.

### 4.3 Phase 3 - Code generation and optimisation

This section considers the improvement (if any) offered by GNORT in terms of code size, quality and understandability over the project’s original Ada83 compiler. Unfortunately, it has not been possible to evaluate the performance of the GNORT-generated system, owing to the unavailability of the SHOLIS target hardware.<sup>2</sup>

**Generated code size.** This is important in embedded systems where ROM and RAM resources may be tightly constrained. Table 1 shows the size in bytes of the code<sup>3</sup> and data sections for SHOLIS, compiled with the Alsys Ada83 compiler, and GNORT at optimisation levels 0, 1 and 2. The code submitted to GNORT was that resulting from the improvements described in section 4.2. The Alsys compiler had all optimisations turned off—the options that were used for delivery of the system. In all four cases, runtime checks were suppressed, and subprogram in-lining was enabled. Here we see that

**Table 1.** Code and Data sizes for Alsys vs. GNORT

Compiler	Code	Data	Total
Alsys	174066	78878	252944
GNORT -O0	202412	71052	273464
GNORT -O1	138280	71052	209332
GNORT -O2	135192	71052	206244

GNORT at optimisation level 0 generates approximately 27k more code than Alsys. This reflects the fact that, with optimisation off, GNAT can achieve the goal of absolutely no optimization at all, since the table driven techniques used can be directed to completely avoid optimization. With a hand written code generator like that used by the Aonix compiler, it is relatively difficult to eliminate *all* optimization of any kind.

At higher levels of optimisation, GNORT does significantly better. In particular, at level 1 the GCC backend implements common subexpression elimination, assigns variables to registers where possible, and eliminates many redundant load and store operations. The code is often shorter, and (perhaps more importantly) makes significantly fewer references to main memory. While it has not been possible to measure the execution time of GNORT’s code on the real SHOLIS hardware, our manual analysis

<sup>2</sup> The prototype SHOLIS system is currently bolted down to a Royal Navy ship, whereabouts unknown!

<sup>3</sup> The code section includes all generated code and constants for the SHOLIS application code. For the Alsys compiler, the SMART runtime system code, constants, and exception tables are not included in this figure.

of several time-critical inner loops suggests that the GNORT code would comfortably out-perform the original system.

**Ability of GNORT to take advantage of SPARK.** As we pointed out in section 3.1, SPARK enforces several static semantic rules (e.g. no function side-effects, no exceptions, staticness of initialising expressions etc.) If a compiler could recognise these properties of a program, then some improvement in code generation may be possible. These features include:

*Exceptions.* SHOLIS is proven to be free of all predefined exceptions. GNORT allows exceptions to be raised, but they always result in a simple jump to a single user-defined routine, which can be used to terminate or restart the program. In a totally exception-free program like SHOLIS, GNORT imposes no overhead—it does not generate unwanted handlers or tables, and does not impose any overhead on the subprogram entry/exit sequence. This contrasts with the Alsys compiler, which imposes no run-time overhead for unused exceptions, but still generates 6095 bytes of exception handling tables in the final program image, which are never used.

*Staticness.* In a SPARK program, constraints are always static, and the expressions initialising constants are always determinable at compile time. As we saw in section 4.2, GNORT does well in this respect. In most cases, it is able to evaluate initialising expressions at compile time, even when the Ada95 definition of “static” does not require it to do so.

*Function side-effects.* SPARK eliminates all function side-effects, which offers some possibility of improved code-generation. Pragma Pure offers one technique for this, and was successfully applied to a small subset of the packages in SHOLIS. GNORT also offers an implementation-defined pragma *Pure\_Function* for this purpose. Experimentation with this pragma did not show any great improvement in generated code, although the algorithms in question simply may not have been amenable to such improvement.

**Optimisation.** In the safety-critical community, “no optimisation” seems the norm. Reasons often cited for this include the perceived unreliability of optimisers in early Ada83 implementations, and the difficulty of debugging and reviewing optimised object code. GNORT (being based on the GCC back-end) is normally run with its optimiser enabled, though, so there may be a strong argument in favour of its use. If object-code integrity is of concern, three approaches can be taken:

- The use of a formally verified compiler.
- Review of the compiler’s in-service history and reliability.
- Manual review of generated code.

The first of these options remains a research topic, and is not an option for a language the size of Ada at present. The second option leads to the question “what compilation options are considered to be most reliable, and therefore most suitable for a safety-critical

system?” For GNORT (and other compilers derived from GCC), informal evidence suggests that `-O1` is the most reliable option, since this option is normally used by the vast majority of users.

Finally, the manual review of object code remains a necessary (but arduous) task for some safety-critical systems. To determine how well GNORT supports such an activity, a single subprogram from SHOLIS was subjected to a manual review, comparing the code generated by the Alsys compiler with that generated by GNORT at optimisation levels 0 and 1. Pragma Reviewable was applied to the unit in question.

The Alsys compiler produces in its listings file details of record type layout, the stack layout for each subprogram, and an annotated disassembly of the unit’s object code. The annotations include source line numbers, the name of objects accessed by any one instruction, and the Ada name of called subprograms. These annotations and stack-layout information significantly simplify the job of reviewing the generated code.

Similar information is available in the GNORT-generated listings. The GNU *objdump* utility can be used to produce a merged listing showing the disassembled code and source. The debugging information can also be reproduced in a human-readable form which reveals the stack layout for each subprogram. At optimisation level 0, the code is as easy to understand as that generated by the Alsys compiler, or better. At level 1, things are slightly more complicated (indeed, pragma Reviewable is not strictly supported at this level.) At this level, the GCC backend can assign objects to registers, and can re-use those registers when the life-times of two such objects do not overlap, so interpreting the code is somewhat more difficult, although not impossible for an experienced user. This increase in difficulty is balanced by the reduction in code volume at this level. Note that at `-O1`, the GCC optimiser does not perform global re-organisation of the code (which *would* complicate any verification considerably.)

A final observation on this topic is that GNAT allows a level of verification not available with most other compilation systems through the availability of the front-end’s intermediate language (IL). The `-gnatD` switch produces the IL as an Ada-like listing for each unit, and places references to the IL in the object code, not to the original source. This implies that the *objdump* listing for a unit then shows the generated code interspersed with the IL listing rather than the original source. This facility offers the prospect of a 2-step verification of object code. The first step would verify the IL against the source code, which requires a detailed knowledge of Ada, but not necessarily of the target machine. The second step would verify the generated code against the IL. This process might ease verification since the “semantic gap” between the languages being compared in each step is far narrower.

## 5 Further work

### 5.1 GNORT

With regard to GNORT, the facilities provided are essentially complete with respect to the SHOLIS requirements. The one technical point is that it is unfortunate that an address clause cannot be written in a pre-elaborable unit, as pointed out in section 4.2. Note that from a code generation point of view, there is no problem here, since the call

to `To_Address` would be inlined and in fact does not generate code at all (it is essentially an unchecked conversion). We propose to add a new implementation defined attribute:

```
for X'Address use Address'Value (...);
```

This special attribute will be defined to have the same semantics as `To_Address`, but will be a static attribute, and thus be allowed in a pre-elaborable unit. This is not strictly a GNORT issue, but for the reasons described in this paper, it is likely to arise in this context. We anticipate similar situations arising in future that may require minor work in the area of additional inlining, or special attributes.

## 5.2 SHOLIS

As for SHOLIS itself, this work has not attempted to take advantage of child packages, which are permitted in SPARK 95. Introducing child packages implies a complete re-think of the program's structure, which was beyond the limits of this investigation, so this remains a topic for future work. Secondly, SHOLIS remains a totally single-task program, with a cyclic scheduler. While this approach offers some benefits in terms of simplicity and efficiency, it has some serious drawbacks: the program's structure is damaged to "fit" computations into the available system cycles, and some real-time requirements are difficult to meet. An obvious development would be to re-work SHOLIS using the Ravenscar tasking profile [4, 6].

## 6 Conclusions

From the findings of these investigations, we can conclude:

- GNORT provides a usable and effective means of producing safety-critical applications, and offers a cost-effective alternative to a traditional "small but certified" run-time.
- Careful use of SPARK 95 features can offer measureable improvements (in terms of code size, complexity, performance) when applied to an existing SPARK 83 application. These improvements can be achieved with relatively little effort.
- With optimisation off, GNORT generates code which is slightly larger than that generated by the project's original Ada83 compiler, but is well suited to debugging and manual review.
- The use of optimisation *can* be justified with a GCC-based compiler like GNORT. This offers significant benefits in terms of code size and performance, without overly detracting from the understandability of the object code (should a manual review be required.)
- GNORT's support for object code review is at least as good as that offered by the original compiler, but also offers the potential for a 2-phase process, based on a program's intermediate language. We note that GNORT's "open source" philosophy makes this style of review possible—an option that may not be available for other proprietary compilers.

## References

1. The procurement of safety-critical Software in Defence Equipment, Ministry of Defence, Interim Defence Standard 00-55 (Parts 1 and 2), Issue 1, April 1991.
2. SPARK—The SPADE Ada Kernel, Edition 3.2, Praxis Critical Systems, October 1996.
3. Alsys Ada Compiler Cross Development Guide for UNIX workstations to Motorola 68k Targets. Version 5.5.2. Alsys Ltd. September 1993.
4. Safety Critical Solutions. Aonix Inc. 1998  
<http://www.aonix.com/Pdfs/CSDS/safecrit/safe.crit.pdf>
5. High Integrity Ada—The SPARK Approach. John Barnes with Praxis Critical Systems Limited. Addison Wesley, 1997.
6. Proceedings of the 8th International Real-Time Ada Workshop: Tasking Profiles. ACM Ada Letters, September 1997.

**Acknowledgements.** The authors would also like to thank Power Magnetics and Electronic Systems Ltd. and the UK MoD for their permission to publish material relating to SHOLIS in this paper. The first author would like to thank the staff of ACT Inc. and ACT Europe for their patience and support during the practical phase of this work.

The principle developers of the SHOLIS at Praxis Critical Systems were: Andy Pryor (technical authority and project manager), Rod Chapman (software development and proof), Jonathan Hammond (formal specification, proof and safety engineering), Janet Barnes (formal specification and proof), and Neil White (program proof).