ASIS–BASED CODE ANALYSIS AUTOMATION

C. DANIEL COOPER

BOEING DEFENSE & SPACE GROUP
P.O.BOX 3707, MS 4A–25
SEATTLE, WA 98124–2207
cdaniel.cooper@boeing.com

The Ada Semantic Interface Specification (ASIS) is an interface between an Ada environment and any tool requiring information from it. An Ada environment comprises valuable semantic and syntactic information that can be exploited for automated code analysis. ASIS provides an open and published callable interface that gives access to this information; further, ASIS has been designed to be independent of underlying Ada environment implementations, thus supporting portability of code analysis tools while relieving tool developers from needing to understand the complexities of an Ada environment's proprietary and evolving internal representation.

(1) BENEFITS OF CODE ANALYSIS

Why is something like ASIS needed? ASIS provides a basis for implementing portable tools that are aimed at analyzing static properties in Ada source code. Such code analysis capability has in general been underleveraged in software development organizations; but for the Ada language in particular, it can greatly enhance the development process. Code analysis automation can harness the excellent software engineering features of Ada to facilitate code comprehension, high reliability, and high quality in the software product. The following text presents some motivational background.

(1.1) DEFINITION

Code analysis is the inspection of software source code for the purpose of extracting information about the software. Such information can pertain to individual software elements (*e.g.,* standards compliance, test coverage), the element attributes (*e.g.,* quality, correctness, size, metrics), and element relationships (*e.g.,* complexity, dependencies, data usage, call trees); thus, it can support documentation generation, code review, maintainability assessment, reverse engineering, and other software development activities.

Extracted information falls into two major categories: *descriptive* reports which present some view of the software without judgement (*e.g.,* dependency trees, call trees), and *proscriptive* reports which look for particular deficiencies — or their absence (*e.g.,* stack overflow, excessive complexity, unintended recursion).

(1.2) APPLICABILITY

Broadly speaking, the application of automated code analysis in the software development process promises, among other things, to:

- promote ***discipline and consistency*** during development, increasing productivity and reducing unintended variation
- provide empirical evidence and ***metrics*** for process monitoring and improvement
- supplement code ***inspection and review,*** diversifying beyond the limitations of testing or manual checking
- preserve ***architectural integrity*** in the software as compromises are made during development
- avoid violations of ***coding standards***, such as the use of inefficient language constructs
- increase the ***correctness and quality*** of delivered software, reducing defects via comprehensive assessment
- enhance ***safety and security*** by applying formal methods to verify assertions made in program code
- expedite ***program comprehension*** during maintenance, for engineers new to the code
- support ***reengineering and reuse*** of legacy code, reducing costs
- result in ***reduced risk*** to budget and schedule

The software development life–cycle phases where code analysis can be beneficially applied include all those in which source code exists: preliminary design (software architecture and interface definition), through testing and system integration, to maintenance and reengineering. Hence, automated code analysis is a technology that primarily supports the back end of the software life cycle.

(1.3) *Motivation*

Over the years, a wide–ranging set of commercial code analysis tools has become increasingly available [3, 11]; examples of such tools include:

- data flow analysis and usage metrics
- invocation (call) trees and cross–reference
- dependency trees and impact analysis
- timing and sizing estimation
- test–case generation and coverage analysis
- usage counts of language constructs
- quality assessment metrics
- coding style and standards compliance
- safety and security verification
- code browsing and navigation
- documentation generation
- reverse engineering and re–engineering
- language translation and code restructuring

Unfortunately, the current state–of–the–practice in software development either omits code analysis support altogether or only incorporates it as an ancillary, undisciplined, *ad hoc* resource. For example, it is not uncommon to find within a given project various home–grown tools that support the above goals but which are not recognized as overtly participating in the development process. Such tools can be quite obtuse (very indirect extraction of information) and are typically incomplete (handling only a subset of the development language). Further, they tend to be project–specific (or even person–specific), and cannot be reused in another project: they are later redeveloped from scratch.

These observations corroborate that the need for code analysis is genuine, and that a common set of uniform tools could provide significant benefits to projects. But in the case of Ada software, commercial code analysis tools have historically proven to be barely adequate, manifesting a variety of problems whose nature and origin are described below.

(2) Technology for Code Analysis

For Ada, why is ASIS the best approach? Code analysis tools are not new, having been available for decades; but the advent of the Ada language has exposed a variety of analysis limitations and has consequently demanded more comprehensive technology. The following text articulates various Ada–specific issues from a historical perspective: it reviews several technologies that have been applied — with varying degrees of success — to code analysis specifically targeted to Ada software. The review is not comprehensive, but it sketches the evolution of issues that have propelled the development of the ASIS concept.

(2.1) *Code Parsers*

Historically, many commercial code analysis tools have been supplied by compiler vendors in conjunction with their compiler products. But as the community of CASE tool vendors has grown, such tools are often available independent of any compiler. Tool developers have found that conventional parser technology is sufficient for most traditional languages; thus when Ada came along, most vendors expected it would suffice to simply adapt their parsers to handle Ada syntax. But for Ada, the result has held many disappointments:

- textual code editors are often sensitive to Ada syntax but not to Ada semantics
- graphical design editors yield valid graphics, but invalid Ada designs
- source–level tools such as debuggers are forced to understand and traverse the internal data structures of program libraries rather than the text of original source files
- reverse engineering and test tools manifest difficulties when trying to resolve overloaded subprogram names or renamings
- except for compilers, Ada tools do not require ACVC certification; hence, such tools typically fail to handle the complete repertoire of Ada language features

Consider the case of a toolsmith who wants to develop a call–tree analyzer. For such a tool to accurately process Ada source files, the toolsmith would be forced to build almost the entire front end of an Ada compiler — a decidedly major undertaking that far out–scopes the original tool building effort. But CASE tool vendors are not in the compiler business; most are reluctant to make this major investment, or have tried and failed. Yet tools built on parser technology alone are not able to fully support the semantic richness of Ada.

(2.2) *DIANA*

Many Ada compilers store program units into libraries. They typically structure the information according to some proprietary internal form, such as trees of DIANA (Descriptive Intermediate Attributed Notation for Ada — note that the following discussion applies to *all* internal forms, but that DIANA is singled out due to its public documentation [4, 5]: DIANA had been intended for standardization, but failed due to the unexpectedly wide variation in internal forms).

Such trees thus encode both syntactic and semantic information about Ada programs. The root of a DIANA tree corresponds to a compilation unit; the nodes correspond to smaller Ada structures, such as declarations and statements. Node attributes may contain descriptive information and references to other nodes.

DIANA trees offer great convenience and power to toolsmiths, and are sufficient to support the implementation of a large variety of tools (besides code generators in compilers). For example, with access to DIANA, the toolsmith who wanted to develop the call–tree analyzer would have a fairly straightforward project. Furthermore, the tool would exhibit better performance, bypassing the needless regeneration (and redundant storage) of intermediate compilation results that are already available in the Ada libraries.

The power of DIANA is sufficient to support the implementation of a virtually unlimited variety of tools. In general, any tool that requires the semantic information generated by an Ada compiler can benefit from access to DIANA. But as with any technology, the use of DIANA also has drawbacks:

- a given implementation of DIANA by a vendor is ***subject to change:*** upgrades can obsolete tools written against previous versions, hampering maintenance
- similarly, DIANA ***implementations vary*** from vendor to vendor: porting a tool across platforms is a risky and costly endeavor
- DIANA is ***hard to use:*** the trees are quite complex, making it difficult to write and debug tools written against a DIANA specification
- the lack of a simple mapping to Ada makes DIANA ***hard to understand:*** as an abstracted representation of an Ada program, it does not map intuitively to Ada constructs
- DIANA is ***not extensible*** by clients; but tools may need to add attributes for storing graphical or other tool–specific information

(2.3) LRM–INTERFACE

To overcome the drawbacks of DIANA while retaining all of its advantages, a growing need arose to make tool development possible at a level higher than the Ada internal representation level. It was these issues that drove some Ada compiler vendors to independently develop proprietary higher–level interfaces that encapsulated their Ada program library implementations.

In particular, Rational developed a product named the LRM–Interface [7] in the late 1980's (it had been intended for internal use, but was found to be commercially viable). It provided nearly the same power as DIANA, through query–oriented services that extracted a variety of information from the internal DIANA trees. The interface was also considerably easier to understand than DIANA, because it used the already–familiar terminology defined in the Ada LRM (the original *Reference Manual for the Ada Programming Language* [13] or its more recent version, the *Ada 95 Reference Manual* [6]). Furthermore, the LRM–Interface was not subject to change (or at least much less so than was the underlying DIANA), so tools written against it could easily migrate to updated implementations.

Regardless of LRM–Interface specifics, this and similar approaches can generally provide great flexibility. While a data structure like DIANA is not user extensible, a functional interface can be extended via user–supplied secondary services built upon the functions already provided. In–house engineers can easily and quickly build *ad hoc* tools, without funding the development or specialization of a third–party commercial tool.

Nonetheless, as might be expected, this approach also has shortcomings:

- importing source code into the tool environment for analysis can require edits that necessarily result in ***code distortion,*** such that original code attributes might not be preserved (*e.g.,* line numbers or the byte sizing of data)
- interface services are ***read–only*** and cannot make annotations in the Ada library or otherwise modify its state
- tools are ***vendor dependent,*** such that a given tool cannot access Ada libraries from multiple vendors, or equivalent tools from multiple vendors cannot access a given Ada library
- data interchange is ***not standardized*** among tools, so users can't configure their own integrated toolsets by choosing from competing or complementary vendors
- within a software engineering environment (SEE), Ada semantic information remains isolated from and ***not integrated*** with other engineering data present in the environment

(2.4) ASIS

Historically, only a few Ada vendors provided access to the information contained in their proprietary Ada program libraries; and each such interface was unique. Thus began to emerge the need for an open standard that would allow vendor–independent, uniform access to that information.

In 1990, the STARS program leveraged some existing informal efforts, and initiated the development of the Ada Semantic Interface Specification (ASIS); but shortly thereafter, the activity became

unfunded due to the STARS decision to no longer support standardization efforts. Despite this, several of the involved vendors (primarily TeleSoft) continued the ASIS work on a volunteer basis. Some time later, Rational also became an active participant, and seeded the draft standard by contributing their LRM–Interface specification to ASIS.

In 1992, the Ada Board recognized the benefits to the Ada community that a standardized ASIS could offer, and recommended that the AJPO director support "by whatever means possible the development of an ASIS standard and its submission to ISO/WG9 for publication." Subsequently, the ACM SIGAda organization launched official sponsorship of this important work, supported though volunteer effort in the ASIS Working Group (ASISWG) [1]. The ASISWG then completed the specification, naming it ASIS 87 (in the United States, ASIS 83) [2]; in December 1993, the AJPO director announced the availability of ASIS and recommended its use. The ASISWG proceeded to develop ASIS for Ada 95.

As the ASISWG has no standardization authority, an ASIS Rapporteur Group (ASISRG) was established in April 1995 by the ISO/IEC JTC1/SC22 WG9, in order to standardize ASIS as an international standard for Ada 95. ASISWG and ASISRG have jointly cooperated to evolve ASIS as an important interface to the Ada 95 compilation environment.

Like its LRM–Interface predecessor, ASIS defines a set of queries and services that provide uniform programmatic access to the semantic and syntactic information contained within Ada libraries (*i.e.,* vendor independence). In addition, for each Ada vendor, ASIS clients are shielded from the evolving proprietary details that implement the vendor's library representations and internal forms (*i.e.,* version independence). ASIS is designed for implementation on a variety of machines and operating systems, while also supporting the Ada semantic requirements of a wide range of client tools.

ASIS services are essentially primitive, intended to support higher level, more sophisticated services that address the varied needs of specialized tools. While ASIS currently operates in a read–only mode, it may eventually be extended to support some (probably limited) update capability, enabling client tools to save application–dependent data (*e.g.,* graphical information) within an existing Ada library. Although an ASIS implementor could readily support read–write features, members of the safety–critical community have emphasized the danger of providing a generalized write capability, since this could enable editing of the internal representation to differ from the original source code.

(3) SUMMARY

The long–term goal is to achieve a critical mass of ASIS implementations. Several are already available, particularly *ASIS–for–GNAT* [8, 9], and more are coming soon. These will promote a new generation of semantically integrated Ada tools, which in turn will increase programmer productivity and product quality, while decreasing Ada development costs. In particular, the availability of ASIS implementations promises to:

- stimulate ***improved quality*** within existing Ada CASE tools; currently, these tend to be weak in supporting full Ada semantics (*e.g.,* in preserving renamed entities, resolving overloaded subprogram names, *etc.*)
- eliminate the need to import Ada source code into secondary Ada compilation environments, resulting in ***no distortion*** or loss in the subject code (*e.g.,* preserving original line numbers and the byte sizing of data)
- enhance ***safety and security*** by providing for a new class of powerful analysis tools that apply formal methods to verify assertions made in source code (*e.g.,* using Pragma Annotate)
- improve the overall ***performance*** of Ada CASE tools, by eliminating the regeneration (and redundant storage) of Ada semantic information that already exists in Ada libraries
- facilitate in–house development of informal but powerful *ad hoc* Ada tools, providing ***flexibility*** as needed without funding Ada CASE vendor specializations
- maximize ***interoperability*** between Ada CASE tools and Ada compilation environments, thus maximizing tool availability
- enable the ***data interchange*** of Ada semantic information between complementary or competing Ada CASE tools, thus maximizing user choices for the best capabilities of each
- promote standardization in software engineering environments, enabling ***data integration*** of Ada semantic information with other engineering data present in the environment
- establish ***enabling technology*** for new Ada CASE tools, by eliminating the need for tool vendor investment in proprietary Ada compiler technology; this will have a major impact on stimulating the development of new code analysis capabilities

(4) REFERENCES

1: ASISWG/ASISRG; Jun95: WWW home page at **`http://www.acm.org/sigada/WG/asiswg`**

2: ASISWG/SIGAda; May94: *Ada Semantic Interface Specification,* ASIS 83/87, version 1.1.1

3: D. Ehrenfried; Jul/Aug94: "Static Analysis of Ada Programs," SIGAda *ACM Ada Letters,* volume XIV #4, pages 28–35

4: G. Goos, W. Wulf, A. Evans, K. Butler; 1983: *DIANA: An Intermediate Language for Ada,* Springer–Verlag LNCS #161

5: Intermetrics, Inc; 1986: *Reference Manual for the Descriptive Intermediate Annotated Notation for Ada (DIANA)*

6: International Standards Organization (ISO), Information technology — Programming languages — Ada; Jan95: *Ada 95 Reference Manual,* standard ANSI/ISO/IEC–8652:1995

7: Rational Software Corp; May89: *Rational Design and Document Support Tools Guide,* document 8043A, version 1.0

8: S. Rybin, A. Strohmeier, E. Zueff; 1995: "ASIS for GNAT: Goals, Problems, and Implementation Strategy," Ada–Europe Symposium Proceedings, Springer–Verlag LNCS #1031, pages 139–151

9: S. Rybin, A. Strohmeier, A. Kuchumov, V. Fofanov; 1996: "ASIS for GNAT: From the Prototype to the Full Implementation," Proceedings of Ada–Europe Conference on Reliable Software Technologies, Springer–Verlag LNCS #1088, pages 298–311

10: Software Productivity Consortium; 1989: *Ada Quality and Style: Guidelines for Professional Programmers,* Van Nostrand Reinhold

11: Software Productivity Consortium; Jun90: "Tools for Static Analysis of Ada Source Code," technical report 90015–N

12: Software Productivity Consortium; Oct95: *Ada 95 Quality and Style: Guidelines for Professional Programmers,* document SPC–94093–CMC, version 01.00.10

13: U.S. Department of Defense, Ada Joint Program Office; Feb83: *Reference Manual for the Ada Programming Language,* standard ANSI/MIL–STD–1815A–1983, ISO/IEC–8652:1987