TASK: UR69
CDRL: 01430
2 November 1990

2

INFORMAL TECHNICAL REPORT

For The

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Intermediate Language Assessment
IRIS/DIANA Analysis

STARS-RC-01430/001/00
Publication No. GR-7670-1158(NP)
2 November 1990

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0002

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

Unisys Defense Systems
Tactical Systems Division
12010 Sunrise Valley Drive
Reston, VA 22091

~~**Distribution Limited to U.S. Government and U.S. Government Contractors only: Administrative (2 November 1990)**~~

TASK: UR69
CDRL: 01430
2 November 1990

INFORMAL TECHNICAL REPORT
For The
SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Intermediate Language Assessment
IRIS/DIANA Analysis

STARS-RC-01430/001/00
Publication No. GR-7670-1158(NP)
2 November 1990

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0002

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

Unisys Defense Systems
Tactical Systems Division
12010 Sunrise Valley Drive
Reston, VA 22091

2 November 1990 STARS-RC-01430/001/00

PREFACE

This document was prepared by Unisys Defense Systems, Valley Forge Operations, in support of the Unisys STARS Prime contract under the STARS Intermediate Language Assessment task (UR69). This CDRL, 01430, is type A005 (Informal Technical Data) and is entitled "IRIS/DIANA Analysis".

This document has been reviewed and approved by the following Unisys personnel:

UR69 Task Manager: Robert G. Munck

Reviewed by: ______________________
Teri F. Payton, System Architect

Approved by: ______________________
Hans W. Polzer, Program Manager

Contents

List of Figures

1 Introduction

An intermediate language (IL) is a language or data form that is used to store information in a convenient way. It is usually a language which is the result of translating from a human-readable form into a form that is "tool-readable," i.e., processable. The archetypical example of an intermediate language is the language a compiler uses to perform semantic checks and generate machine code.

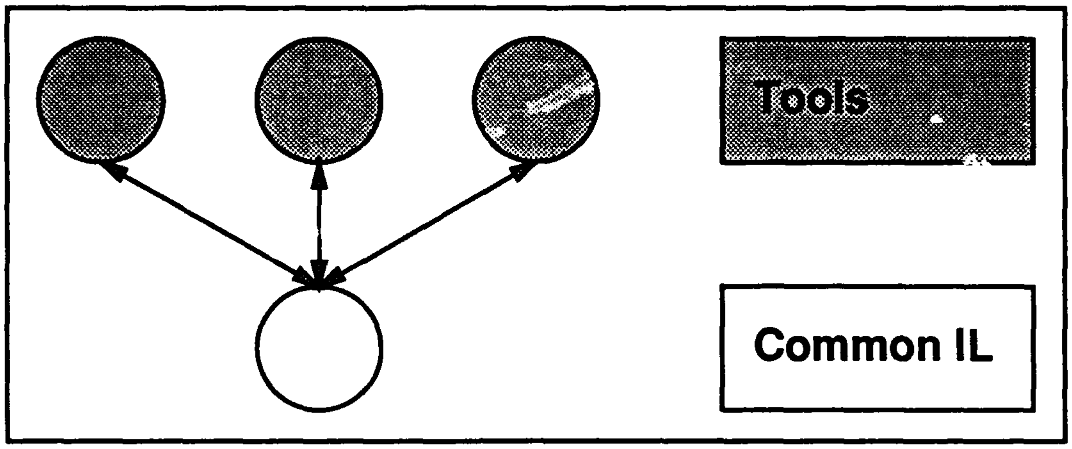
In recent years, intermediate languages have been identified as a partial means for close integration of tool suites, and at present most compiler vendors use a single intermediate form for all their semantic-based tools. Figure 1 shows the normal configuration of these tool suites. Each tool can read necessary portions of the IL and calculate its needed information. This information can be calculated once and stored in the IL as attributes or calculated each time it is requested, depending on time and space tradeoffs. It is important to understand that semantic based tools that use an IL are not limited to compilers and related tools. Currently, editors and verification systems are being constructed with detailed knowledge of the semantics contained in programs, and in the near future tool builders and environment users will discover new and innovative uses for ILs to perform tasks that are today performed without an IL.

Nevertheless, difficulties have arisen through the use of common ILs. The single common IL found in the tool suites described above are usually proprietary and have no open programmatic access to the IL, except through the functionality provided by the tool suite. This provides the benefits of the common IL only to vendors' development and prevents these benefits in customers' development. To counter the trend towards proprietary ILs, the STARS project has fostered the development of several non-proprietary ILs and interfaces, e.g., DIANA by Peregrine Systems and Iris by Incremental Systems, and plans to fund an effort by TeleSoft (the Ada Semantic Interface Specification—ASIS) to develop an open interface to various proprietary ILs. By funding several ILs STARS can experiment with a multi-IL environment, as well as explore different strategies for implementing a single common IL.

The use of different ILs on STARS itself has led to a schism in tool development, as well as difficulty in integrating any tool based on a vendor's proprietary IL. Figure 2 represents the current state of IL usage in STARS. Several tools have been written using DIANA as their IL; several tools have been written using Iris; and other third-party tools are based on a proprietary IL. A translator written by Peregrine Systems converts DIANA into Iris, so that it is possible to integrate tools that use DIANA with those that use Iris, but the translation is only one-way, so it is not yet possible to integrate Iris tools with those that use DIANA.

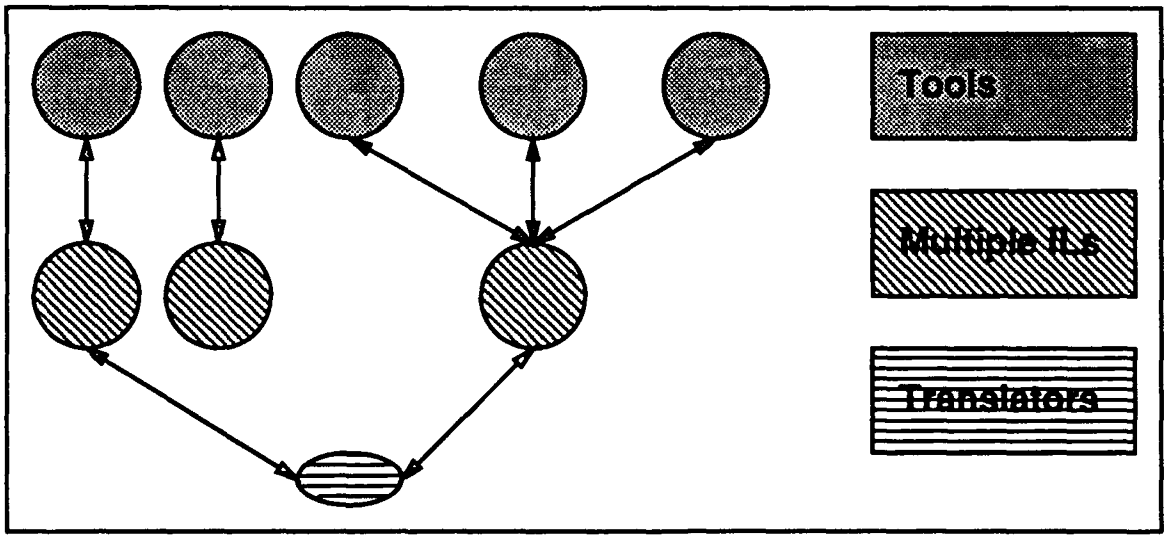
The next step for STARS is to develop a strategy of use for ILs that eliminates the problems of different ILs (some proprietary), and will allow all tools based on ILs to exchange data freely. There are several approaches to this strategy:

1. Mandated single IL
2. Common functional interface to various ILs



A single common intermediate language (IL) standardizes the format of semantic-based information, and allows the free exchange of semantic-based information between tools.

Figure 1: A Single Common IL



The use of multiple intermediate languages prevents the free exchange of semantic-based information between tools. Many tool vendors use proprietary intermediate languages as a means for ``locking in'' a customer base.

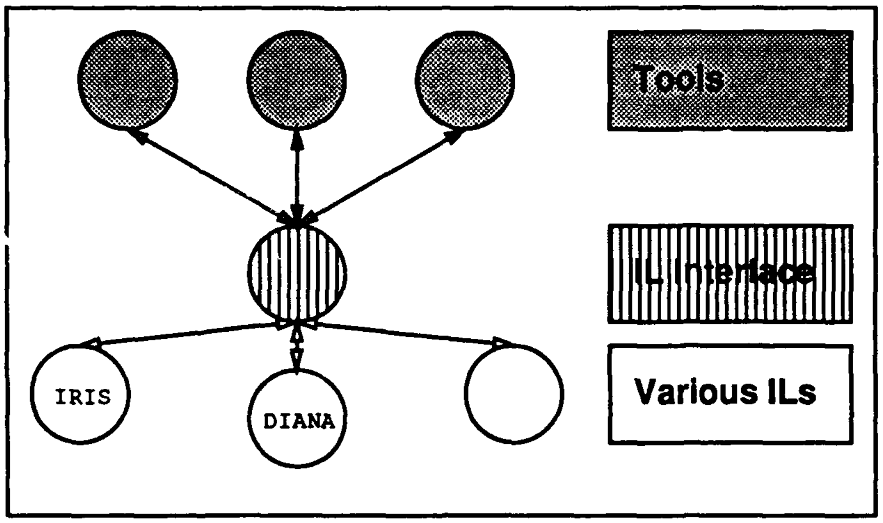
Figure 2: Multiple ILs

3. Multiple ILs with translation functions
4. Hybrid—single IL and functional interface with translation functions

Mandated Single IL

The first solution is a generalization of the successful use of a common IL for tool suite developers, as represented in Figure 1, except that tools in the environment are written by different companies. If every STARS environment tool that is based, however loosely, on semantic information of the Ada language used the same IL, semantic information could be exchanged between these tools without the difficulty of incompatible data formats (assuming that the information that is exchanged is contained in the attribute of the common IL).

However, there are several drawbacks in scaling this solution to the environment level for both open and proprietary tools. Conversion of all open tools written using ILs other than the selected IL must be done, before they can be integrated into the environment. Third party tools may not want to participate because of the effort involved and the openness it forces on their product line. Nevertheless, a single common IL is a goal that STARS should strive to achieve, with the understanding that variations need to be explored that minimize these drawbacks.



A functional interface allows many different intermediate languages to co-exist in an environment. However, it de-emphasizes the different features of intermediate languages, such as, IRIS's capability to represent incorrect and incomplete programs.

Figure 3: Functional Interface to Multiple ILs

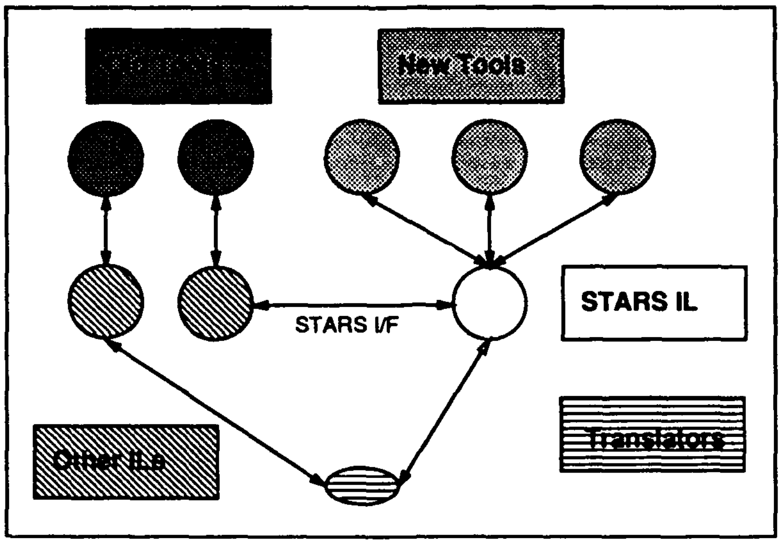
Common Functional Interface

This solution (see Figure 3) rises from the practical problems of competing tool vendors cooperating by selecting a common IL for all semantic based tools. It has the benefit of allowing tool vendors to maintain their current IL, and not have to re-write their tools.

Technical questions arise as to exactly what ought to be abstracted from underlying ILs. Should an interface provide access to everything necessary for semantic based tools to operate unrestricted? How is this different than choosing one of the existing ILs as the standard and forcing all other ILs to provide the selected IL's interface as an alternative to their own?

Multiple ILs with Translation

An alternative to a common functional interface is the reverse, i.e., translation from every IL to every other IL. The benefit of this approach is that no existing tool needs to be re-implemented, but this solution is costly, and not a good long-term solution. Since all new third party vendors will be required to implement two-way translators for every IL pre-existing in the environment, as the environment grows and the number of ILs increases, the cost of introducing a new tool based on a different IL rises.



By choosing a single common intermediate language the STARS environment can gain the same benefits tool vendors gain through the use of a single intermediate language. However, certain evolutionary paths, i.e., translators and funtional interfaces, must be established for existing tools to migrate to the standard intermediate language.

Figure 4: Evolution to Common STARS IL

Hybrid

Through the use of a hybrid approach (see Figure 4), the benefits of the above approaches can be emphasized and the weaknesses de-emphasized. By selecting a single common IL and its interface as the standard IL and interface, all tools implemented after the selection can be based on the common IL, and achieve the benefit of the first alternative. However, to counter the weakness of mandating a single IL, the mandate should not apply to previously constructed tools. These tools have two possible ways to be integrated into the environment. The first is to provide the same interface as the mandated IL to their IL. The second is to create a translator from their IL to the mandated IL.

Using this approach, as existing tools mature they will evolve through various stages to use the common IL, and they will not be excluded from the environment during their evolution. The cost of writing one translator from an IL to the common IL is not prohibitive, only the possibility of writing many translators is prohibitive. This approach limits the maximum number of translators necessary to the number of ILs currently existing in the environment. Also, it is not likely that the maximum number of translators will be necessary, since tool vendors may choose to implement the common IL interface instead of a translator.

By choosing a single common intermediate language the STARS environment can gain the same benefits tool vendors gain, i.e., free exchange of semantic based information, and by using a hybrid approach, evolutionary paths are left open for tools that are currently not based on the selected IL. This approach would be the most fruitful for STARS to follow.

2 Background of STARS ILs

In order to select an appropriate IL as the STARS standard, it is important to understand the context and status of each IL. The following sections detail the history, philosophy, and status of each IL that is being investigated by STARS.

2.1 History

DIANA

DIANA (Descriptive Intermediate Attributed Notation for Ada) is a synthesis of two earlier intermediate forms for Ada programs:

- AIDA—developed at the University of Karlsruehe
- TCOL/Ada—developed at Carnegie-Mellon University and Intermetrics

The actual design of DIANA was conducted by teams from Karlsruehe, Carnegie-Mellon, Intermetrics and SofTech, at a workshop at Eglin Air Force Base which was organized for this purpose. The goal was to combine the best features of these two existing intermediate forms and to encourage other compiler-development and tool-development efforts to adopt a common representation.

A Reference Manual describing DIANA was widely disseminated in 1983[3]. This manual describes the method of defining DIANA using Interface Description Language (IDL), a notation for describing the structure of attributed trees. The manual includes a definition of DIANA as an abstract data type, using the IDL notation; it also includes a discussion of the rationale for the design of DIANA.

A revision of the DIANA specification was developed under AJPO auspices and published in 1986[2]. The important modifications include:

- Overhauling the representation of types and subtypes to better represent the Ada concept of subtypes.
- "Partitioning" the DIANA so that classes and nodes form a strict hierarchy.
- Attaching attributes to the highest appropriate classes rather than to individual nodes.

The modifications made in this revision solve a number of problems in the 1983 version. For this reason, the 1986 version has been used in the STARS implementation of DIANA.

Iris

IRIS (Intermediate Representation Including Semantics was originated by Incremental Systems Corporation. Iris was used in the Incremental Systems Ada compiler beginning in December 1984 and was presented at an Arcadia consortium meeting held in December 1984. Iris concepts and a grammar and set of specifications were further refined and used in an Ada compiler effort finished in December 1988. A standard Iris-Ada instantiation was agreed upon for the Arcadia consortium in October 1989.

ASIS

Ada Semantics Interface Specification (ASIS) is a programmatic interface to high-level services for supporting static semantic analysis of Ada objects. It is being developed by TeleSoft. The specification is being developed with the hope of eventually being adopted as a standard to which compiler vendors could provide implementations. The purpose of ASIS is to allow tools access to selected high-level parts of a compilation system's intermediate language in order to provide a means for their integration. Several levels of abstraction are proposed: a high-level interface to Ada library units, a high-level interface to tool services, and a low-level interface to the complete static semantics represented by an IL.

2.2 Philosophy

Each IL has several underlying assumptions. Understanding these assumptions can provide insight into how the particular features of the IL will affect tool development and the overall STARS environment.

DIANA

DIANA was designed for the purpose of representing Ada programs in a way which allows efficient and straightforward implementation of:

- The communication between the front end and the back end of a compiler, and
- Tools which analyze or manipulate Ada programs.

Some of the design principles of DIANA which support the above requirements are:

- DIANA can be easily extended. Additional nodes or attributes can be added by inserting them in the IDL description; IDL processing tools then make the extensions available to DIANA users.

- DIANA can be efficiently implemented. The STARS implementation is built on a binary representation which provides a front end with processing times comparable to commercial compilers. The library system is built around fast binary input-output of DIANA structures.
- DIANA is representation independent. DIANA is defined as an abstract data type; the implementation of that data type is not constrained. The details of the representation are hidden from the user of the Ada interface.

The IDL description of DIANA as a hierarchy of nodes and classes, with specific named attributes being valid for specific nodes and classes, permits substantial checking of programs using DIANA prior to runtime. This may be done either through an interface in which the different nodes and classes correspond to different Ada types, or through an analysis tool that checks consistency of attribute usage. (The STARS implementation adopts the latter choice.)

Iris

There are two main parts of the Iris representation: the Iris form and an extensible attribute system. The Iris form represents a program (written in some formal language) as a composition of applications and references. The Iris form is in turn represented as an extensible, attributed, information structure consisting of collections of entities and their associated attributes.

The Iris form is tree-structured with only two kinds of nodes: reference and application. Each Iris tree represents an expression composed of applications and references. Reference nodes are interpreted as references to declarations that appear elsewhere within the Iris structure. Application nodes are interpreted as the application of an operation to a sequence of arguments. The operation is always the value of the operator (which is the leftmost subtree) of the application node. The arguments are the values of the remaining subtrees. If the reference nodes of Iris are viewed as leaves (terminals), then the Iris representation can also be viewed as an abstract syntax tree with the application nodes acting as nonterminals. Each reference node, however, contains a reference to a declaration which is itself an application node appearing earlier in (a preorder walk of) the Iris structure.

Iris is unique in that all operations are described within its own structure. This means that individual tools need to recognize and provide special case processing for only those operations that relate directly to the functionality of the tool. For example, the overload resolution portion of a semantic analyzer needs to recognize only those operations that are declaration, scope, or type valued. It does not have to distinguish between control structures and arithmetic operations. This means that individual tools and tool components are often significantly smaller than with traditional representations. Also, because tools process most operations based on the internal definition of the operation rather than by explicit reference, the language being represented can often be modified or extended without modification to the tools.

Iris is also a higher order system in that it provides full support for computed operations at any level. A computed operation may appear either in place at the point of its application (i.e., as another application node which is the operator of the application) or as the value of a declaration which is referenced at the point of call (i.e., as a reference node which is the operator of the application). The combination of internal and higher order specification means Iris can be used to represent any formal language and that Iris based tools can be reconfigured for multiple languages and other changing requirements with little or no change to their components.

The Iris form can represent many incomplete and incorrect programs in addition to correct programs. Any composition (nesting) of language constructs has a corresponding Iris tree. Therefore, tools can be applied to programs represented in Iris throughout their development, even before they have been completely written and made semantically correct.

The Iris form is represented as an extensible information structure so that it can be extended to satisfy the needs of particular tools and applications. The Iris form is in turn represented in terms of an entity-relationship-attribute (ERA) model, where relationships are represented as attributes whose values are "pointers" to other entities. The basic entities of Iris-Ada are nodes, which represent the nodes of the Iris form; tokens, which represent the lexical elements of programs such as identifiers, operators, and literals; comments, which represent comments appearing in the program source; and errors, which are used to identify errors in the represented program.

Tools can create new kinds of entities, or add attributes to existing entities without affecting other tools that use those entities. Therefore, Iris can represent information about programs in addition to the programs themselves, and share this information with other tools. As environments evolve, it is expected that some of these additional attributes will be standardized as well, in order to promote the sharing of information among tools. However, it will never be possible to standardize a set of a[illegible]ibutes that is sufficient for all tools and so it is important that an internal representation be open-ended in order to accommodate new requirements as they arise.

ASIS

The design effort behind ASIS is to provide as simple an interface as possible for tools to gather static semantic information about Ada units. A large part of the effort is to eliminate the appearance of the tree data structures that are currently used in many intermediate languages. In addition, an effort is being made to identify the level of abstraction of an intermediate language that would hide most of the intermediate language's details, while still allowing tools to gather static semantic information.

The design philosophy of ASIS is driven more by what a tool might want to ask of the intermediate language, rather than by computing the information itself using a direct interface to the intermediate language. For example, the ASIS interface provides calls to return all compilation units that *with* a particular compilation unit, return all the compilation units that a particular compilation unit *withs*, return the tasks having a particular entry, or return the subprograms having a particular formal parameter.

2.3 General Status

DIANA

A number of compilers use a variant of DIANA as their internal form. These have started with the public definition of DIANA, normally DIANA 83, but have diverged because there has not been a lot of pressure for conformance. In addition, they generally have compiler-specific, proprietary interfaces. (One of the goals of the STARS development was to make available a public domain version in the hope that if a critical mass of tools based on one common DIANA were available, there would then be some real pressure for conformity and interoperability.)

The defining methodology for DIANA, IDL, has been further developed apart from its use in describing DIANA. Most of this work has no strong connection with Ada. A special issue of SIGPLAN Notices[4] was devoted to papers from a workshop on IDL sponsored by the University of North Carolina and the SEI.

Iris

Iris has been in use at Incremental Systems and other members of the Arcadia consortium since 1984. It has continued to evolve since then and as a result several different versions of Iris are in use in Arcadia. In October, 1989, a draft standard Iris-Ada instantiation was agreed on for Arcadia.[1] Subsequently, several problems with the draft standard have been found and corrections have been made. This process will continue in the near term as Arcadia tools are converted to the standard and additional problems are discovered.

ASIS

The ASIS interface is currently in the design phase. The evaluation of ASIS in this report is based on a release of ASIS that is not available to the general public, and, as of the writing of this report, no general release of ASIS has been made.

2.4 STARS Status

DIANA

A prototype implementation of DIANA and associated tools was developed for STARS by Peregrine Systems, Inc., under subcontracts with the Institute for Defense Analyses and CEA, Inc. The prototype has been used on a number of projects both within STARS and outside of STARS since December, 1988. A productized implementation, correcting deficiencies in the prototype and bringing it up to the level where it is accepted by Version 1.11 of the ACVC, will be delivered in September, 1990. Both of the above versions are based on the 1986 version of DIANA.

DIANA prototype version

The prototype version consists of the following components:

- A definition of DIANA in IDL form. The definition is the same as that given in the Intermetrics report[2] with a few minor corrections and with a small number of additional nodes and attributes intended for use within the Ada-to-DIANA front end.
- An Ada binding for DIANA, i.e., an Ada package which allows creation and manipulation of DIANA trees from an Ada program. Two bindings ; provided: One consists of a package which looks very much like the Ada binding in the 1983 DIANA report. In this binding, attributes are represented as functions for obtaining given attributes of a node and procedures for storing such attributes. Another binding, which was less stressful for smaller Ada compilers in use at the start of the project, represents attributes as an enumeration type and has a single function and a single procedure to obtain and store an attribute in a node. (Either binding may be used; different compilation units within a compilation are not required to use the same one.)
- An Ada-to-DIANA front end, which accepts legal Ada source code and produces the corresponding DIANA. This tool accepts legal Ada and constructs the corresponding DIANA. The entire Ada language is accepted; the product accepts all of the Class A and Class C tests of Version 1.10 of the ACVC test suite. At the prototype level, there was no attempt to do a complete test for input which is not legal Ada, although most checks are done; in consequence, the Class B tests are not all rejected as invalid, as required by the test suite. (Approximately 30 tests of ACVC Version 1.11 are not accepted; these will not be corrected in the prototype version.)
- An Ada library, as part of the Ada-to-DIANA front end, to provide for separate compilation and inter-unit checking.
- A tool for pretty-printing DIANA trees. This tool provides an indented listing showing the values of all attributes of each node.
- A facility for loading DIANA trees for previously-translated compilation units from the Ada library. (A program intended to do further manipulation of DIANA trees can use this facility to obtain access to the trees.)
- A tool to read the IDL description of DIANA and generate the Ada bindings described above.

- An LALR parser generator and an Ada grammar, used to produce the parse phase of the Ada-to-DIANA front end, including the descriptions of the DIANA trees to be created by the parser.

DIANA productized version

The productized version contains the same components as described above, plus a "Linker" which checks that a set of compilation units can be combined into an Ada program. The linker reads the DIANA tree for each compilation unit required by a particular Ada main program and constructs a tree for the program; this permits the ACVC Class D tests to be run.

The productized version is correct, as tested by Version 1.11 of the ACVC, including the Class A, Class B, Class C, Class D and Class L tests. (Note that manual checking of the generated DIANA trees for correctness is prohibitive; it is intended by STARS to perform this check by comparing the results of DIANA overload resolution with the results of overload resolution in the Incremental Systems Iris front end.)

The IDL definition of DIANA is the same for the productized version as for the prototype version, except that the additional nodes and attributes intended for use by the front end have been modified. The Ada bindings should be upward compatible, provided that these front-end-specific elements have not been used. Subtypes of the DIANA type TREE have been provided so that the uses of attributes and nodes can be checked prior to run time; a tool is provided to do this check. (An attempt to provide such checking purely via Ada strong type checking was abandoned because the required number of types and functions stressed existing Ada compilers so much that they could not be used.)

Projects Using STARS DIANA

The following is a list of projects which are known to be currently using or to have used the STARS implementation of DIANA.

- Unisys—Verification Condition Generator
- Harris—STARS prototype Ada test tools
- Grumman—Database tools (Eugene Vasilescu)
- Grumman—Static analysis tools for Ada
- University of Texas
- GTE—Implementation of Ada metrics tools

Iris

Iris was first introduced into the Unisys STARS program in mid 1989. Since that time, Incremental Systems has written an Ada to Iris translator and the basic set of components needed to create, access, and manipulate the Iris representation. This implementation is based on a version of the front end of the Incremental Systems Ada compiler written in Pascal. For STARS, this front end was translated to Ada and some design improvements were made. It was also updated to the Arcadia standard as it existed in April 1990; in future increments, the translator will be modified to reflect any changes to the standard occurring after that date.

Currently, to date, Odyssey Research Associates has been using Iris for Ada Formal Methods work. Incremental Systems is working with Unisys Valley Forge Laboratories to convert the Ada Command Environment (ACE) to use Iris. The intent is for "light-weight" verification tools (fragments) to work from an extended Iris-Ada representation.

ASIS

Development efforts for prototype implementation of initial ASIS bindings are being initiated under STARS in late 1990.

It is planned for several tools (both COTS and public domain) to be integrated making use of the ASIS. STARS will also experiment using ASIS as a high-level programmable interface across several compilation systems, including the Iris to Ada translator.

3 Intermediate Languages

This section describes the merits of each intermediate language as an intermediate language, i.e., for the purposes of communicating semantic-based information between tools. See the appendix for example code that illustrates the use of DIANA and Iris.

3.1 Intended Use

DIANA

DIANA is intended to be used to represent programs in the Ada language. Its intended use is as an internal form for communication between the front end and the back end of an Ada compiler or for communication with tools which analyze or synthesize Ada programs.

DIANA can be extended to hold additional information which may be required by various tools which use it. The basic definition of DIANA includes only those nodes and attributes which are required to describe the syntax of an Ada program or to represent semantic information which would be difficult for an individual tool to compute. (In the design of DIANA, semantic information was generally considered not difficult to compute if it could be obtained either by a single walk through the tree at tool invocation or by visiting only a small number of nodes in the tree.)

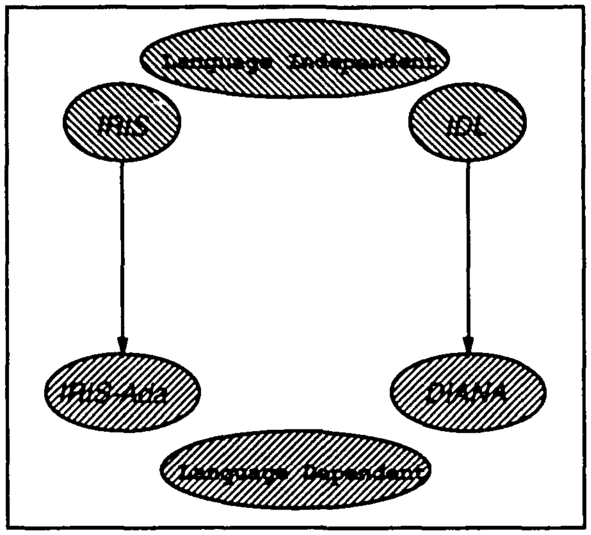
DIANA is defined using a methodology called Interface Description Language (IDL) which allows the nodes and attributes of DIANA to be declared in a succinct way. The IDL methodology can equally be used to define similar representations for languages other than Ada. (Thus, IDL corresponds to "language-independent" Iris, while DIANA corresponds to the "language-specific" Iris-Ada.)

Iris

Iris is intended to be used to represent "programs" in any formal language. It can be used by tool suites that analyze, query, and synthesize programs. Because Iris is language-independent, tools can be written which process programs in several languages. Because Iris is open-ended, it can be extended to represent information about programs in addition to the programs themselves. Therefore, Iris can be a key integration mechanism in a software environment.

ASIS

ASIS does not represent an IL per se. It is a high level programmatic interface that hides the often complicated structures used in most intermediate languages. The abstraction of the intermediate language it presents is constrained by the range of languages it will abstract. For example, the ability of Iris and DIANA to represent incorrect programs is not presented



IRIS-Ada and DIANA are generated from high-level specification languages. These specification languages allow for the creation of many language dependent systems, which can be used as a family of intermediate languages in a multi-lingual environment.

Figure 5: General Method of Creation for DIANA and Iris

or even a factor in the ASIS interface. The interface can only present the information stored in the IL, not the techniques used to create the information.

On the other hand, ASIS will represent a basic set of queries that most tools will ask of ILs. The query based abstractions can be represented in the attributes of DIANA and Iris, but would have to be calculated by the tools themselves, unless a general tool calculated them for all tools. Having this interface generally available would remove the need for tools to calculate them on an ad hoc basis.

3.2 Method of Use

DIANA

DIANA is described by its IDL description. The latter gives all node types and their attributes. A binding to Ada which matches this description (and which was created by a program which reads IDL descriptions) is used for constructing programs which manipulate the IDL tree.

An Ada binding was given in the DIANA 83 document. No binding was given in the DIANA 86 document-the appropriate section was left blank; therefore a binding similar to the DIANA 83 version has been adopted for the STARS implementation.

In the STARS implementation, a tool will typically attach itself to the software virtual memory which contains the DIANA data structure, either a "brand new" virtual memory or one passed from another tool. It can then import existing DIANA trees corresponding to various compilation units, as required, using functions provided. It will then walk the tree (or trees) using a recursive tree-walk algorithm to visit each node.

Following operation of a tool which creates or modifies DIANA trees, the resulting trees can be written out. The collection of files containing existing DIANA trees corresponds to the Ada Reference Manual's concept of an Ada library.

Where necessary for construction of a tool, the DIANA definition must be extended by modifying the IDL description to include the required additional nodes and attributes.

Iris

For Iris-Ada, the starting point is the Ada grammar, the Ada augmented declarations, and the basic entities and attributes of Iris. For some tools, this is adequate because these give the tool access to the representation of any Ada program.

Many tools, however, will collect or compute additional information about programs. For such tools, the tool writer may find it convenient to store that information as additional entities or attributes. Using the attribute manager, the tool writer can extend the representation to include these additional entities and attributes and to provide operations for them that can be used by tool writers to access them from other tools.

This view should encourage tool writers to build small, composible tool fragments that compute, collect, or access a particular kind of information which can be shared among several tool fragments. Because the Iris representation is easily extensible, this information can be represented in a uniform manner thus promoting this kind of sharing.

For the majority of tools, this kind of extensibility is adequate. For a few tools, the tool writer will need to extend or modify the language represented. For example, the Ada language can be extended by supplementing it with annotation languages that are represented as structured Ada comments. The tool writer must then extend the set of augmented declarations to include the predefined semantic operations of the annotation language. They must also write a grammar which specifies the relationships between the syntactic constructs of the language and their representation in the Iris abstract syntax. And finally, specific tools must be extended to "know" the semantic details of the annotation language. Note, however, that the attributes of the annotation language are invisible to tools designed to work on unannotated Ada programs and therefore, such tools can continue to be used with annotated programs.

ASIS

The use of ASIS is the same as any Ada interface. Package specifications represent the types and operations that can be used, hiding the underlying structure from the user. There is no attribute system, but the plans for ASIS include operations that are normally associated with attributes. Therefore, to extend the "attributes" of ASIS, the interface itself would need to be extended.

3.3 Problems and Issues

DIANA

A number of compiler vendors purport to use DIANA as the internal form for their compilers. In general, these vendors have taken DIANA (usually DIANA 83) as a starting point to define an intermediate representation; the resulting representation is similar to DIANA, but there seems to be little standardization between vendors. In addition, the representation is frequently considered to be proprietary, and a programmatic interface to it is not publically available, except if a source license for the particular compiler is purchased. The result is that tools which make use of the compiler's internal representation cannot be built without a source license, cannot be distributed without a license from the compiler vendor, and can only be used with a single compiler. A major purpose in building a STARS implementation of DIANA was to provide a basis for a non-proprietary DIANA, with the hope that translations to and from Ada compiler intermediate languages could be added as its use grows.

In both the prototype and the productized versions of STARS DIANA, adding an attribute for use by a particular tool requires that the attribute be added in the IDL definition of DIANA and all users be recompiled. This is clearly undesirable, and could be largely remedied by providing an appropriate implementation of refinements for IDL definitions.

Iris

The basic Iris-Ada representation was designed to be minimal in that redundant information is not stored. Therefore, any information which is computable from other stored information, even though that computation may be expensive, is not stored.

An example of this appears in the sample tool accompanying this report (see Appendix). In Ada, when a derived type is declared, some subprograms that have parameters or return values of the parent type of that derivation are derived; that is, a subprogram implicitly declares all occurrences of the parent type replaced by the derived type. In Iris-Ada, these implicit declarations are not represented and so all references to the derived subprograms are resolved to refer to the explicitly-declared ancestor.

Many tools do not need to distinguish between the explicit subprogram and any subprograms derived from it. Storing that information, whether by adding the implicit declarations to the tree or by storing additional attributes, would make all such tools pay a storage penalty to store unneeded information. On the other hand, some tools do need this information. Because it is expensive to compute it will probably become one of the standard "auxiliary" attributes, i.e., not part of the "minimal" Iris-Ada core set but one which can be loaded by tools when it is needed.

ASIS

There are no concrete problems with ASIS because the interface is still being defined and currently has no implementation. However, a loss of generality would be expected, since attributes or queries for all possible tools cannot be enumerated *a priori*, and *a posteriori* enumeration of attributes would require recompilation of all ASIS based tools, if they are to remain consistent.

4 Comparison

DIANA and Iris are very similar. There are only a few technical factors on which the two can be compared. ASIS, however, is a completely different style than DIANA or Iris, and, therefore, cannot be compared in the same way. For example, both Iris and DIANA have the capability of representing incorrect programs to various degrees. ASIS, however, cannot make any assumption on the ability of the underlying IL to represent incorrect programs, since it is an interface in principle to many ILs and is constrained to the most common elements of all ILs.

4.1 Level of Abstraction

The main difference between ASIS and more detailed interfaces, such as those used by DIANA and Iris, is the level of abstraction supported by the interface. ASIS is concerned with high-level queries on Ada objects. For example, a compilation unit security model will be investigated using the level of abstraction provided by ASIS. However, since ASIS provides a very high-level of abstraction there are many tasks for which it would not be suited without access to the complete IL and an extensible attribute system. For example, ASIS in its present form does not provide enough information to be used as the intermediate language for two existing Unisys STARS tools: the Ada Command Environment (ACE) and the Ada verification system, Penelope. ASIS could be extended for these tools by adding the necessary queries to the ASIS interface, but without an extensible attribute system, the ASIS interface will grow as each new tool requires a different set of attributes. One could argue that each tool should manage the attributes it uses that are not represented in ASIS, i.e., the attributes it calculates, but then we are restricting the usefulness of a common IL by not allowing the free exchange of information derived from the IL using IL attributes.

The implied use of ASIS is as a supplement to the compilation system, and not as an interface to the entire intermediate language. If ASIS will address access to the entire IL then there is a question as to why is redundant work being performed, and the reasons why the Iris interface or the DIANA interface is not an appropriate starting point for such work should be enumerated. Also, there are questions as to whether or not the level of access to the IL provided by ASIS will ever evolve to the detail that DIANA and Iris interfaces represent. Since ASIS is being proposed as an interface to many different intermediate languages, for it to be supported by all compiler vendors as a standard, ASIS must be able to be implemented efficiently on a vendor's intermediate language. The more fine grained the access to the information contained in an intermediate language, the less chance of particular compiler vendors being able to efficiently implement the interface.

4.2 External Attribute Extension

The greatest benefit DIANA and Iris provide over functional interfaces is the ability to define new attributes that the IL will manage. To add an operation (i.e, attribute) to an interface can be a costly operation, since every tool using that interface will need to be recompiled in order to remain consistent. And if the interface is a standard, standard approval must be sought for the operation addition, which might be more difficult than working around the missing operation. A major difficulty with a standard, functional interface is the inherent lack of extensibility.

Iris and DIANA both allow the IL that they represent to be extended via external attribute definitions. Iris currently has a distinct advantage over DIANA in that the definition of new attributes does not require all tools based on Iris to be recompiled. This allows an Iris-based environment to evolve without affecting existing tools.

4.3 Precomputed Attributes

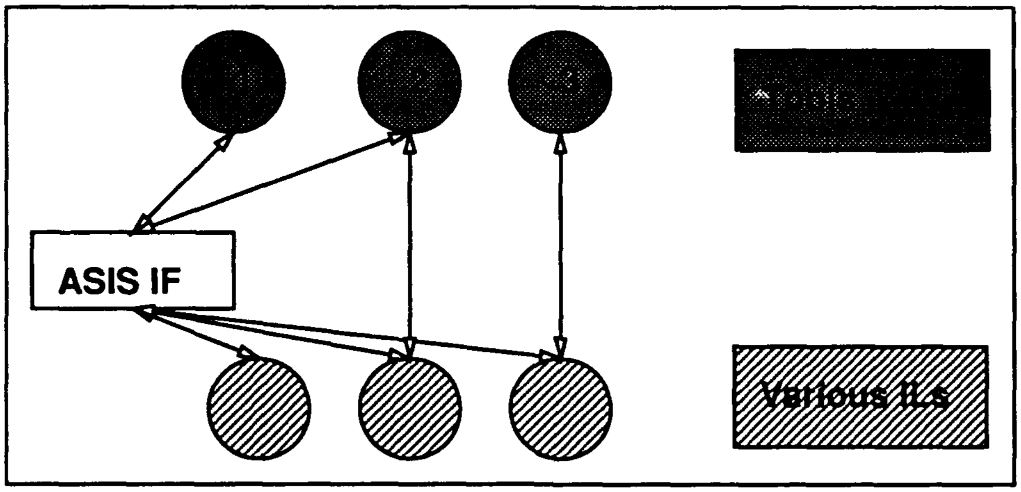
The standard Iris structure does not include information which may be difficult to compute, except for the resolutions to the (underived) declaration corresponding to references to defined names. For example, a tool may need to know which of several visible derived functions (with a common parent) is referred to by a reference. In this case, DIANA 86 points directly to a *function_id* representing the derived function; Iris identifies only the common parent and a repetition of a substantial part of visibility analysis may be required to determine the specific derived function. However, the correction for this problem is to identify a set of standard attributes that will be implemented (akin to the ASIS interface) and available to all tool writers.

4.4 Represenation of Incomplete Programs

Both Iris and DIANA have the ability to represent incorrect programs. DIANA can represent semantically incorrect programs, by allowing particular attributes in the abstract syntax tree to remain undefined. Iris can also represent the same class of semantically incorrect programs. However, Iris also has the ability to represent a class of syntactically incorrect programs. The class of syntactically incorrect programs is limited to nested structures. For example, Iris can easily handle programs with missing **begin** statements, since that implies a list of nested declarations and statements, but it cannot deal well with missing semicolons.

4.5 Special Purpose vs. Generality

DIANA is a special purpose intermediate language that was defined for tools that are specific to Ada compilation systems, while Iris is a general intermediate language that is ultimately intended for a multi-lingual environment. Because DIANA is described with specific attributes for specific purposes, the values of those attributes can be checked for validity, thus helping to ensure the integrity of the DIANA tree. The same sort of checking is more difficult in Iris, precisely because of its generality and its ability to represent incorrect programs.



ASIS will provide a high-level interface to allow tools to query the underlying intermediate language without understanding the structure of the intermediate language itself. It remains an open question as to how much of the intermediate language can be abstracted without loosing critical information and forcing tools to access the intermediate language directly.

Figure 6: ASIS Interface

4.6 Functional Interfaces

ASIS (Ada Semantic Interface Specification) will be a common functional interface to different underlying intermediate languages. The effort in ASIS is to provide a high-level set of operations that most tools use. These high-level operations can be thought of as the attributes of the underlying intermediate language, and would be a *first cut* at the standard set of attributes that the STARS IL should implement.

However, a functional interface hides much about the nature of the underlying IL, such as the ability to process incorrect programs, that can be critical to the operation of tools. For example, suppose a tool is written using the ASIS interface, with Iris as the underlying IL, for the purpose of processing incorrect programs. The same tools using an ASIS interface to the Verdix Ada compiler's intermediate language would not work correctly, since the Verdix IL does not represent incorrect programs.

There are several approaches to using the ASIS interface in the STARS environment. Figure 6 represents three classes of tools with respect to the ASIS interface. The first tool (1) is totally isolated from the IL by ASIS, and will port to any IL that provides the ASIS interface (albeit with some restriction because of the nature of the underlying IL as described above). The second tool (2), while it uses the ASIS interface, needs more detailed information, and must access the IL directly. The ASIS interface provides a convenient set of operation for some of the tasks performed by the tool, but not all. The third tool (3) performs all low-level operation so that no access to the ASIS interface is necessary or desired.

The most important feature of including ASIS into the STARS environment is the access it provides to the previously proprietary intermediate languages used by Ada compiler vendors. While it does not solve all the problems of integrating third party tools and having all STARS tools operate from a single interface, it does provide the only access to the underlying structure of commercial Ada compilation systems. Therefore, while there are difficulties in using ASIS for a complete IL, it serves a very necessary function in the STARS environment.

5 Recommendations

It is recommended that a hybrid approach to implementing a single common STARS intermediate language be used. The ASIS interface should contribute the complete STARS IL interface, and Iris should be selected as the STARS IL.

It is not recommended that STARS focus solely on the ASIS specification for its intermediate language needs because of its high-level of abstraction and lack of extensibility, making it unsuitable for many STARS tasks, e.g., ACE and Ada Formal Methods. and Penelope). However, the ASIS interface should be a component of the STARS interface to the STARS IL providing a common high-level interface to the STARS IL and commercial off-the-shelf Ada compilation systems. This implies that the standardization efforts for ASIS should also be pursued by STARS.

Iris is recommended over DIANA as the STARS IL. The primary reason is the functionality of Iris that separates attributes from the structure of the intermediate language allowing tools to add attributes without effecting other Iris-based tools in the environment. Selection of Iris will also allow the products of the Arcadia consortium to be easily integrated into the STARS environment.

References

[1] Incremental Systems Corporation. *Reference Manual for Iris*, 1990. Technical Report 891001.

[2] Intermetrics Inc. *Diana Reference Manual*, 1986. IR-MD-078, Draft Revision 4.

[3] Tartan Laboratories Inc. *Diana Reference Manual*, 1983. TL-83-4, Draft Revision 3.

[4] SIGPLAN Notices, November 1987. Vol. 22, No. 11.

Appendix

The following two sections are real examples of the use of DIANA and Iris. They are provided for the reader to gain an understanding of the interfaces produced, and the complexity involved in using the ILs for special purpose tools.

The following examples are tools which will analyze whether the context clauses of a compilation unit are necessary or not. This is a particularly interesting problem to Ada coders, since it is often not known whether or not a context clause is necessary without commenting out the context clause and submitting the code to the Ada compiler for verification. The code for these tools is relatively short and was written in a few hours.

DIANA

```
-- The following is an example of the use of DIANA in a tool which
-- produces a list of units required to be with'ed by a given compilation
-- unit (irrespective of the list given in the context clause)

-- A unit must be with'ed if name declared within it is used in one
-- of the following two contexts:
--        (1)  As a directly visible name in the given unit
--        (2)  As a default actual generic parameter, where the
--             instantiation occurs in the given unit and the
--             corresponding formal parameter default is "is <>".

-- To keep the size of the example down, the with clauses required by
-- the following are omitted:
--        (a)  Implicitly defined operators (These require consideration of
--             sufficient context to determine the unit containing the
--             declaration of the type for which an operator is declared.)
--        (b)  Default actual generic parameters for formals with box
--             defaults.  (This is case (2) above.)

-- Also, to keep the size of the example down, the required withs are
-- simply printed when found, rather than collected.

-- Note that a unit will be listed for a subunit even though a with
-- is given for it in the corresponding library unit or in an ancestor
-- subunit.

--================================================================

package Chkwith is
```

```
   -- This is the main package for the example tool.  The tool is invoked
   -- by a call to one of the procedures below.  The required unit
   -- must have been processed by the DIANA front end and be present
   -- in the DIANA library.

   procedure Check_Withs_For_Spec (Library_Name : String);

   procedure Check_Withs_For_Body (Library_Name : String);

   procedure Check_Withs_For_Subunit (Library_Name : String;
                                      Subunit_Name : String);

end Chkwith;

--================================================================

with Userpk;
use Userpk;

package Srchwth is

   -- The following procedure is called from CHKWITH to perform the
   -- actual search after the given unit has been loaded from the
   -- DIANA library.  STANDARD_ID is the package_id node for
   -- predefined STANDARD; LIBRARY_ID is the package_id, function_id,
   -- procedure_id or generic_id of the compilation unit in which
   -- the given unit is declared.

   procedure Search_All_Withs (Comp_Unit   : Tree;
                               Standard_Id : Tree;
                               Library_Id  : Tree);

end Srchwth;

--================================================================

with Userpk;
use Userpk;

package Srchutl is

   -- This package provides various small routines used by the search
   -- process.  It provides for recording library units as they are
   -- seen and for finding the type structure corresponding to
   -- different type marks.  (The latter is needed to determine
   -- if an aggregate is a record aggregate, in which case the names
   -- in named associations are not directly visible names.)

   -- defining id of predefined STANDARD
   -- (set from package SRCHWITH)
   Standard_Id : Tree;

   -- defining id of library unit of given unit
   -- (set from package SRCHWITH)
   Library_Id  : Tree;

   -- given a defining id (package_id, entry_id, ...)
   -- ... return the defining id of the library unit in which
   -- ... the given id is declared
   function Library_Unit_Of (Id : Tree) return Tree;

   -- procedure to check if library unit already seen
   -- ... and, if not, remember that it was seen and print message
   procedure Record_Library_Unit (Id : Tree);

   -- procedure to clear the set of units that have been seen
   procedure Clear_Units_Seen;

   -- given a node in class TYPE_SPEC, return the base type of the
   -- ... full type spec (i.e., look inside private and l_private)
   function Get_Base_Struct (Type_Spec : Tree) return Tree;

end Srchutl;

--=============================================================

with Userpk;
use Userpk;
with Set_Simple_Sequential_Unbounded_Unmanaged_Iterator;

package Srchset is new

-- This package provides a facility for manipulating sets
-- of library-unit identifiers.  (A generic set package
-- is assumed to be available.)

   Set_Simple_Sequential_Unbounded_Unmanaged_Iterator (Item => Tree);
```

```
--==============================================================
-- **** BEGIN PACKAGE BODIES ****
--==============================================================

with Text_Io;
use Text_Io;
with Userpk;
use Userpk;
with Diana86;
use Diana86;
with Unitlod;
with Srchwth;
with Srchutl;

package body Chkwith is

   -- This is the main package for the example tool.  The tool is invoked
   -- by a call to one of the exported procedures. The required unit
   -- must have been processed by the DIANA front end and be present
   -- in the DIANA library.

   procedure Check_Withs_For_Comp_Unit (Comp_Unit : Tree);

   -- **** EXPORTED SUBPROGRAMS ****

   -- these procedures create a new DIANA virtual memory,
   -- ... load the required unit from the DIANA library
   -- ... and call a common subprogram to do the actual searching

   -- Note.  Package UNITLOD is provided with DIANA and contains procedures
   -- to obtain access to DIANA trees for units in the DIANA library.

   procedure Check_Withs_For_Spec (Library_Name : String) is
   begin
      Create_DIANA;
      Check_Withs_For_Comp_Unit (Unitlod.Load_Library_Spec (Library_Name));
      Close_DIANA;
   end Check_Withs_For_Spec;

   procedure Check_Withs_For_Body (Library_Name : String) is
   begin
      Create_DIANA;
      Check_Withs_For_Comp_Unit (Unitlod.Load_Library_Body (Library_Name));
      Close_DIANA;
   end Check_Withs_For_Body;
```

```
procedure Check_Withs_For_Subunit (Library_Name : String;
                                   Subunit_Name : String) is
begin
   Create_DIANA;
   Check_Withs_For_Comp_Unit (Unitlod.Load_Library_Subunit (Library_Name,
                                                             Subunit_Name));
   Close_DIANA;
end Check_Withs_For_Subunit;

-- **** INTERNAL SUBPROGRAMS ****

-- common procedure called by the three exported subprograms
-- to search one compilation unit for required with clauses

procedure Check_Withs_For_Comp_Unit (Comp_Unit : Tree) is

   -- the compilation unit containing predefined STANDARD
   Predefined_Unit          : Tree;

   -- intermediate nodes used in finding standard_id
   Predefined_Package_Decl : Tree;
   Predefined_Package_Spec : Tree;
   Standard_Package_Decl   : Tree;

   -- package_id for predefined standard
   Standard_Id              : Tree;

   -- id for library unit of given compilation unit
   Library_Id               : Tree;
begin

   -- locate the unit containing predefined STANDARD
   Predefined_Unit := Unitlod.Load_Library_Spec ("_STANDRD");

   -- if the unit was not found in the library
   if Comp_Unit = Const_Void then

      -- print error
      Put_Line ("**** Unit not found in library");

      -- else if predefined STANDARD was not found
   elsif Predefined_Unit = Const_Void then

      -- print error
```

```
        Put_Line ("**** Predefined STANDARD not found in library");

        -- else -- since the unit and predefined STANDARD were found
      else

        -- find the package_id for predefined standard
        -- it's the name in the first declaration, a package declaration,
        -- ... in the visible part of the (dummy) package _STANDRD
        Predefined_Package_Decl := As_All_Decl (Predefined_Unit);
        Predefined_Package_Spec := As_Header (Predefined_Package_Decl);
        Standard_Package_Decl :=
          Head (List (As_Decl_S1 (Predefined_Package_Spec)));
        Standard_Id := As_Source_Name (Standard_Package_Decl);

        -- find the id for the region containing the unit to be searched
        if Kind (Comp_Unit) = Dn_Subunit then
          Library_Id :=
            Srchutl.Library_Unit_Of
               (Sm_First
                  (As_Source_Name
                     (As_Subunit_Body (As_All_Decl (Comp_Unit)))));
        else
          Library_Id := Sm_First (As_Source_Name (As_All_Decl (Comp_Unit)));
        end if;

        -- find and print required withs
        Put_Line ("Searching for required withs...");
        Srchwth.Search_All_Withs (Comp_Unit, Standard_Id, Library_Id);
        Put_Line ("Search complete.");
      end if;
    end Check_Withs_For_Comp_Unit;

end Chkwith;

--================================================================

with Diana86;
use Diana86;
with Srchutl;

package body Srchwth is

    -- The procedure SEARCH_ALL_WITHS is called from CHKWITH to perform
    -- the actual search after the given unit has been loaded from the
    -- DIANA library.  STANDARD_ID is the package_id node for
-- predefined STANDARD; LIBRARY_ID is the package_id, function_id,
-- procedure_id or generic_id of the compilation unit in which
-- the given unit is declared.

procedure Search_Withs (T : Tree);

procedure Search_Withs_In_Assoc_S (T : Tree);

-- **** EXPORTED SUBPROGRAMS ****

procedure Search_All_Withs (Comp_Unit   : Tree;
                            Standard_Id : Tree;
                            Library_Id  : Tree) is
begin

   -- remember standard and library ids
   Srchutl.Standard_Id := Standard_Id;
   Srchutl.Library_Id  := Library_Id;

   -- clear the set of library units seen so far
   Srchutl.Clear_Units_Seen;

   -- call recursive tree walk to search for withs
   Search_Withs (Comp_Unit);

end Search_All_Withs;

-- **** INTERNAL SUBPROGRAMS ****

-- Given a piece of the DIANA tree for the given compilation unit,
-- ... search (recursively) for library units which must be withed.
-- (This procedure is the heart of the example; it does the recursive
-- walk throught the syntax tree.)

procedure Search_Withs (T : Tree) is
begin

   case Kind (T) is

      when Dn_Pragma   =>

         -- if this is not an ignored pragma (i.e., pragma_id not void)
         if Sm_Defn (As_Used_Name_Id (T)) /= Const_Void then

            -- walk expression arguments
            -- note that sm_defn will be void for a used_..._id
            -- ... which is not a reference
            Search_Withs_In_Assoc_S (As_General_Assoc_S (T));
         end if;

      when Class_Call_Stm   =>

         -- don't search left side of argument association
         Search_Withs (As_Name (T));
         Search_Withs_In_Assoc_S (As_General_Assoc_S (T));

      when Class_Designator =>
         declare
            The_Defn : Tree := Sm_Defn (T);
         begin
            -- if this is not a defined name
            if The_Defn = Const_Void then

               -- do nothing
               null;

               -- else if this is a builtin operator
            elsif Kind (The_Defn) = Dn_Bltn_Operator_Id then

               -- **** BUILTIN OPERATORS IGNORED ***
               null;

               -- else -- since this is defined name and not builtin
            else

               -- add to list of library units
               -- (print name if this is the first time seen)
               Srchutl.Record_Library_Unit
                   (Srchutl.Library_Unit_Of (Sm_Defn (T)));
            end if;
         end;

      when Dn_Selected =>

         -- don't search designator
         Search_Withs (As_Name (T));

      when Dn_Function_Call =>

         -- don't search left side of argument association
```

```
            Search_Withs (As_Name (T));
            Search_Withs_In_Assoc_S (As_General_Assoc_S (T));

         when Dn_Aggregate     =>

            -- if this is a record aggregate
            if Kind (Srchutl.Get_Base_Struct (Sm_Exp_Type (T))) = Dn_Record
                then

               -- don't search the left sides of associations
               Search_Withs_In_Assoc_S (As_General_Assoc_S (T));

               -- else -- since this is an array aggregate or subaggregate
            else

               -- treat normally (left sides of => are expressions)
               Search_Withs (As_General_Assoc_S (T));
            end if;

         when Dn_Dscrmt_Constraint  =>

            -- don't search left side of argument association
            Search_Withs_In_Assoc_S (As_General_Assoc_S (T));

         when Dn_Instantiation =>

            -- **** BOX DEFAULT ACTUALS IGNORED ***
            -- don't search left side of argument association
            Search_Withs (As_Name (T));
            Search_Withs_In_Assoc_S (As_General_Assoc_S (T));

         when Dn_Compilation_Unit    =>

            -- don't search the context clause
            Search_Withs (As_All_Decl (T));
            Search_Withs (As_Pragma_S (T));

         when others =>

            -- for all node kinds which do not have special treatment
            -- search all subnodes in the syntax tree
            case Arity (T) is
               when Nullary    =>
                  null;
               when Unary =>
```

```
                  Search_Withs (Son1 (T));
               when Binary     =>
                  Search_Withs (Son1 (T));
                  Search_Withs (Son2 (T));
               when Ternary    =>
                  Search_Withs (Son1 (T));
                  Search_Withs (Son2 (T));
                  Search_Withs (Son3 (T));
               when Arbitrary  =>
                  declare
                     Item_List : Seq_Type := List (T);
                     Item      : Tree;
                  begin
                     while not Is_Empty (Item_List) loop
                        Pop (Item_List, Item);
                        Search_Withs (Item);
                     end loop;
                  end;
            end case;
      end case;
   end Search_Withs;

   ------------------------------------------------------------

   -- Search for withs required by expressions in a sequence of
   -- parameter associations or in a sequence of named associations
   -- in a record aggregate.  (In these cases, the name to the
   -- left of the "=>" in the association is not searched as it
   -- is not a directly-visible reference.)

   procedure Search_Withs_In_Assoc_S (T : Tree) is
      Item_List : Seq_Type := List (T);
      Item      : Tree;
   begin
      while not Is_Empty (Item_List) loop
         Pop (Item_List, Item);
         if Kind (Item) in Class_Named_Assoc then
            Search_Withs (As_Exp (T));
         else
            Search_Withs (T);
         end if;
      end loop;
   end Search_Withs_In_Assoc_S;

end Srchwth;
```

```
--==================================================================

with Text_Io;
use Text_Io;
with Diana86;
use Diana86;
with Srchset;

package body Srchutl is

   -- the set of library units seen so far
   Library_Units_Seen : Srchset.Set;

   -- **** EXPORTED SUBPROGRAMS ****

   -- given a defining id (package_id, entry_id, ...)
   -- ... return the defining id of the library unit in which
   -- ... the given id is declared

   function Library_Unit_Of (Id : Tree) return Tree is
      Region : Tree;
   begin

      -- if this is STANDARD, denoting predefined STANDARD
      if Id = Standard_Id then

         -- consider it as its own library unit
         return Id;

         -- else -- since it is a name other than predefined STANDARD
      else

         -- get enclosing region
         Region := Xd_Region (Id);

         -- if the enclosing region is predefined STANDARD
         if Region = Standard_Id then

            -- the given id is a library unit or defined in STANDARD
            if Kind (Id) in Class_Unit_Name then
               return Id;
            else
               return Standard_Id;
            end if;
         return Id;

         -- else -- since the enclosing region was not STANDARD
       else

         -- search recursively for the library unit of that region
         return Library_Unit_Of (Region);
       end if;
    end if;
 end Library_Unit_Of;

 --------------------------------------------------------------

 -- procedure to check if library unit already seen
 -- ... and, if not, add to LIBRARY_UNITS_SEEN and print message

 procedure Record_Library_Unit (Id : Tree) is
 begin

    -- if the unit is predefined STANDARD or contains the given unit
    if Id = Standard_Id or Id = Library_Id then

       -- ignore it
       null;

       -- else if the unit has not yet been seen
    elsif not Srchset.Is_A_Member (Id,
                                   Of_The_Set => Library_Units_Seen) then

       -- print message
       Put ("... requires: with ");
       Put_Line (Printname (Lx_Symrep (Id)));

       -- add to set of library units seen
       Srchset.Add (Id,
                    To_The_Set => Library_Units_Seen);
    end if;
 end Record_Library_Unit;

 --------------------------------------------------------------

 -- procedure to make the set LIBRARY_UNITS_SEEN empty

 procedure Clear_Units_Seen is
 begin
```

```
      Srchset.Clear (Library_Units_Seen);
   end Clear_Units_Seen;

   ------------------------------------------------------------

   -- given a node in class TYPE_SPEC, return the base type of the
   -- ... full type spec (i.e., look inside private and l_private)

   function Get_Base_Struct (Type_Spec : Tree) return Tree is
   begin
      if Kind (Type_Spec) not in Class_Type_Spec then
         return Type_Spec;
      end if;

      case Class_Type_Spec'(Kind (Type_Spec)) is
         when Dn_Task_Spec      =>
            return Type_Spec;
         when Class_Non_Task    =>
            if Type_Spec = Sm_Base_Type (Type_Spec) then
               return Type_Spec;
            else
               return Get_Base_Struct (Sm_Base_Type (Type_Spec));
            end if;
         when Class_Private_Spec     =>
            return Get_Base_Struct (Sm_Base_Type (Type_Spec));
         when Dn_Incomplete     =>
            return Get_Base_Struct (Xd_Full_Type_Spec (Type_Spec));
         when Dn_Universal_Integer |
              Dn_Universal_Fixed   |
              Dn_Universal_Real      =>
            return Type_Spec;
      end case;
   end Get_Base_Struct;

end Srchutl;
```

Iris

```
--
-- Copyright 1990, Incremental Systems Corporation.
--
--     Permission is granted to reproduce this material provided
--     that (i) such copies are not made or distributed for
--     commercial advantage, (ii) this copyright notice appears
--     on each whole or partial copy, and (iii) any modified or
--     partial copies are clearly identified as such.
--
--                       Disclaimer
--
--     This software and its documentation are provided "AS IS"
--     and without any expressed or implied warranties
--     whatsoever.  No warranties as to performance,
--     merchantability, or fitness for a particular purpose
--     exist.
--
--     In no event shall any person or organization of people be
--     held responsible for any direct, indirect, or
--     consequential or inconsequential damages or lost profits.
--
--     STARS R-Increment Preliminary Delivery  (June 7, 1990)
--
with Node_Entity;
use Node_Entity;
with Segment_Definitions;
with Library_Manager;
with Primitives;
use Primitives;
package Node_With_Level_Attr is
   package Sd renames Segment_Definitions;
   package Ne renames Node_Entity;
   package Lm renames Library_Manager;

   Node_With_Level_Attribute : Lm.Attribute;

   type With_Level is  range 0 .. 2 ** 15 - 1;

   subtype Node_With_Level_Body is With_Level;

   type Node_With_Level is access Node_With_Level_Body;

   Null_Node_With_Level      : constant Node_With_Level := null;

   procedure Crea e_Node_With_Level_Attr (U             : in Lm.Unit;
                                          Item_Count    : in Nat32 := 1;
                                          Growth_Factor : in Nat32 :=
      Sd.Default_Growth_Factor);

   -- Allocate a With_Level attribute of node t
   -- Set the initial value to zero
   procedure Allocate (T    : in Node;
                       R_Wl : out Node_With_Level);

   procedure Deallocate_Node_With_Level (T : in Node);

   function Fetch (T : Node) return Node_With_Level;
end Node_With_Level_Attr;
```

```
--
-- Copyright 1990, Incremental Systems Corporation.
--
--     Permission is granted to reproduce this material provided
--     that (i) such copies are not made or distributed for
--     commercial advantage, (ii) this copyright notice appears
--     on each whole or partial copy, and (iii) any modified or
--     partial copies are clearly identified as such.
--
--                       Disclaimer
--
--     This software and its documentation are provided "AS IS"
--     and without any expressed or implied warranties
--     whatsoever.  No warranties as to performance,
--     merchantability, or fitness for a particular purpose
--     exist.
--
--     In no event shall any person or organization of people be
--     held responsible for any direct, indirect, or
--     consequential or inconsequential damages or lost profits.
--
--     STARS R-Increment Preliminary Delivery  (June 7, 1990)
--
with Heterogeneous_Segment_Definitions;
with Attribute_Labels;
package body Node_With_Level_Attr is

   package Node_Segment_Definitions is new Heterogeneous_Segment_Definitions
       (Index => Nonnull_Node_Index);

   package Nsd renames Node_Segment_Definitions;
   package Al renames Attribute_Labels;

   procedure Create_Node_With_Level_Attr (U             : in Lm.Unit;
                                          Item_Count    : in Nat32 := 1;
                                          Growth_Factor : in Nat32 :=
      Sd.Default_Growth_Factor) is
      S : constant Sd.Segment :=
         Sd.Segment
             (Nsd.Create_Segment (Item_Count, Node_With_Level_Body'SIZE,
                                  Growth_Factor, Al.Attribute_Label'SIZE));
   begin
      Al.Label_Segment (S);
      Lm.Assign_Attribute_Segment (U, Node_With_Level_Attribute, S);
   end Create_Node_With_Level_Attr;
   procedure Node_With_Level_Allocate_With_Init is new Nsd.Allocate_With_Init
       (Node_With_Level_Body, Node_With_Level);

   procedure Allocate (T    : in Node;
                       R_Wl : out Node_With_Level) is
   begin
      Node_With_Level_Allocate_With_Init
           (Nsd.Heterogeneous_Segment
                (Lm.Attribute_Segment (T.U, Node_With_Level_Attribute)), T.N, 0,
            R_Wl);
   end Allocate;

   procedure Deallocate_Node_With_Level (T : in Node) is
   begin
      Nsd.Deallocate
           (Nsd.Heterogeneous_Segment
                (Lm.Attribute_Segment (T.U, Node_With_Level_Attribute)), T.N);
   end Deallocate_Node_With_Level;

   function Node_With_Level_Fetch is new Nsd.Fetch (Node_With_Level_Body,
                                                    Node_With_Level);

   function Fetch (T : Node) return Node_With_Level is
   begin
      return
         Node_With_Level_Fetch
              (Nsd.Heterogeneous_Segment
                   (Lm.Attribute_Segment (T.U, Node_With_Level_Attribute)), T.N);
   end Fetch;

begin
   Lm.Define_Attribute_Kind (Node_With_Level_Attribute, "node_with_level");
end Node_With_Level_Attr;
```

```
--
-- Copyright 1990, Incremental Systems Corporation.
--
--     Permission is granted to reproduce this material provided
--     that (i) such copies are not made or distributed for
--     commercial advantage, (ii) this copyright notice appears
--     on each whole or partial copy, and (iii) any modified or
--     partial copies are clearly identified as such.
--
--                      Disclaimer
--
--     This software and its documentation are provided "AS IS"
--     and without any expressed or implied warranties
--     whatsoever.  No warranties as to performance,
--     merchantability, or fitness for a particular purpose
--     exist.
--
--     In no event shall any person or organization of people be
--     held responsible for any direct, indirect, or
--     consequential or inconsequential damages or lost profits.
--
--     STARS R-Increment Preliminary Delivery  (June 7, 1990)
--
with Command_Option;
with Context_Constructor;
with Driver_Command_Information;
with Driver_Environment;
with Error_Messages;
with Io_Exceptions;
with Library_Manager;
with Node_Entity;
use Node_Entity;
with Node_Form_Attr;
use Node_Form_Attr;
with Node_With_Level_Attr;
with Segment_Definitions;
with Text_Io;
use Text_Io;
with Token_Entity;
with Standard_Decls;
use Standard_Decls;
procedure With_Check is

   package Cc renames Context_Constructor;
   package Co renames Command_Option;
package Dci renames Driver_Command_Information;
package De renames Driver_Environment;
package Em renames Error_Messages;
package Lm renames Library_Manager;
package Nwla renames Node_With_Level_Attr;
package Sd renames Segment_Definitions;
package Te renames Token_Entity;

type String_Ptr is access String;

type Option is  range 0 .. 1;

O_First      : constant Option := 0;
O_Library    : constant Option := 1;

type Option_Entry is access String;

type Option_Table is array (Option range <>) of Option_Entry;

Options      : constant Option_Table := (O_First   => new String'(""),
                                         O_Library => new String'("-library"));

Library_Name : String_Ptr; -- name of library containing unit
-- to be processed
Lib          : Lm.Library; -- handle to current library
Unit_Name    : String_Ptr; -- unit to be processed
Errors       : Natural; -- number of context construction
-- errors

function Key (I : Option) return String;
procedure Find is new Command_Option.Find_Option (Option, Key);

function Key (I : Option) return String is
begin
   return Options (I).all;
end Key;

procedure Interpret_Invoking_Command is
   I : Integer;
begin
   Dci.Program := new String'(De.Arg_Value (0));

   I            := 1;
   while I <= De.Arg_Number loop
      declare
```

```
            This_Option : Option;
            Opt         : constant String := De.Arg_Value (I);
            Result      : Co.Search_Result;
         begin
            exit  when Opt (1) /= '-';
            Find (Opt, This_Option, Result);
            case Result is
               when Co.Search_Exact | Co.Search_Prefix  =>
                  case This_Option is
                     when O_Library =>
                        I            := I + 1;
                        Library_Name := new String'(De.Arg_Value (I));
                     when O_First   =>
                        Em.Internal_Error ("first option returned");
                  end case;
               when Co.Search_Ambiguous  =>
                  Em.Fatal_Error ("ambiguous option " & Opt);
               when Co.Search_Not_Found  =>
                  Em.Fatal_Error ("unrecognized option " & Opt);
            end case;
         end;
         I := I + 1;
      end loop;

      if I > De.Arg_Number then
         Em.Fatal_Error ("no unit specified");
      else
         Unit_Name := new String'(De.Arg_Value (I));
      end if;

      if Library_Name = null then
         begin
            Library_Name := new String'(De.Environment ("ADA_LIBRARY"));
         exception
            when De.No_Such_Variable     =>
               Library_Name := new String'("library");
         end;
      end if;
   end Interpret_Invoking_Command;

   procedure Load_Attr (Un   : Lm.Unit;
                        Attr : Lm.Attribute) is
      --
      -- Loads an attribute collection
      --
      -- Prints an error message and terminates the tool if an attribute
      -- cannot be loaded.
   begin
      Lm.Load_Attribute (Un, Attr);
   exception
      when Lm.Cannot_Load =>
         Em.Fatal_Error ("cannot load attribute " & Lm.Name_Of (Attr) &
                         " of unit " & Lm.Name_Of (Un));
   end Load_Attr;

   function Root_Of (U : Lm.Unit) return Node is
      --
      -- Returns the real root of the Iris tree for library unit u.
      --
   begin
      return Op (Fetch ((U, 1)));
   end Root_Of;

   procedure Init_Context_Attr (Un : Lm.Unit) is
      --
      -- Loads the token_image attributes and creates the
      -- node_with_level attributes of unit un.  Also allocates a single
      -- node_with_level attribute for the root of node of unit un.  The
      -- attribute manager initializes this attribute to zero when it
      -- is created.
      --
      -- Note: the node_form and local_unit_unit attributes are loaded by the
      -- context constructor.
      --
      T_Wl : Nwla.Node_With_Level;
   begin
      if Lm."=" (Lm.Grammar_Unit, Un) then
         return ;
      end if;

      Load_Attr (Un, Lm.Token_Image_Attribute);

      -- create the with level attribute collection
      Nwla.Create_Node_With_Level_Attr (Un);

      -- Allocate a single attribute on the root of the tree
      Nwla.Allocate (Root_Of (Un), T_Wl);
   end Init_Context_Attr;

   --
```

```
-- Creates and loads the required attributes for all loaded units.
--
procedure Initialize_Context_Attributes is new Lm.For_All_Loaded_Units
    (Init_Context_Attr);


--
-- Sets the node_with_level attribute of the root node of all compilation
-- units containing declarations referred to by the unit rooted at node
-- t to level, if the value of that attribute is not already greater than
-- level.
--
procedure Mark_Referenced_Units (T     : Node;
                                 Level : Nwla.Node_With_Level_Body) is
   procedure Mark (T : Node) is
      T_F  : Node_Form;
      D    : Node;
      D_Wl : Nwla.Node_With_Level;
   begin
      if "=" (T, Null_Node) then
         Put_Line ("<<null>>");
      else
         begin
            T_F := Fetch (T);
            if Is_Application (T_F) then
               for I in 0 .. Num_Args (T_F) loop
                  Mark (Arg (T_F, I));
               end loop;
            elsif Te."=" (Lit_Kind (T_F), Te.Resolved) then -- resolved
               D := Referent (T_F);
               D_Wl := Nwla.Fetch (Root_Of (D.U));
               if Nwla."<" (D_Wl.all, Level) then
                  D_Wl.all := Level;
               end if;
            else -- unresolved
               null;
            end if;
         exception
            when Sd.No_Such_Association    =>
               Put_Line ("<<<missing>>>");
         end;
      end if;
   end Mark;
begin
   Mark (Arg (Fetch (T), 3));
end Mark_Referenced_Units;
```

```
procedure Mark_Unit_And_Ancestors is
   --
   -- Sets the context node_with_level attributes based on the references
   -- of the primary unit and all of its ancestors.  The primary unit
   -- is assigned level 1 and its ancestors are assigned sucessive levels
   -- numbering back from the primary unit.  The language unit is not
   -- processed because it refers to declarations in no unit other than
   -- itself.
   --
   -- When this procedure has completed, node_with_level attribute of
   -- each unit in the context is set to the level of outermost ancestor
   -- unit that references it.
   --
   procedure Mark (R : Node;
                   L : Nwla.Node_With_Level_Body) is
      R_F : constant Node_Form := Fetch (R);
   begin
      if Referent (Fetch (Op (R_F))) /= N_Language_Spec then
         -- first recursively process the ancestor units
         Mark (Referent (Fetch (Arg (R_F, 1))), Nwla."+" (L, 1));
         -- then process all references in this unit
         Mark_Referenced_Units (R, L);
      end if;
   end Mark;
begin
   Mark (Root_Of (Lm.Primary_Unit), 1);
end Mark_Unit_And_Ancestors;

function Name_Of (L : Nwla.Node_With_Level_Body) return Lm.Unit_Name is
   --
   -- Returns the name of the library unit at level l.  It goes
   -- down the ancestor chain
   -- l steps and then calls the library manager function Name_Of
   -- to get the name of the unit found there.
   --
   N : Node := Root_Of (Lm.Primary_Unit);
begin
   for I in 2 .. L loop
      N := Arg (Fetch (N), 1);
   end loop;
   return Lm.Name_Of (N.U);
end Name_Of;

procedure Check_Unit_And_Ancestors is
```

```
--
-- Checks the context clauses of the primary unit and all of its
-- ancestors.
-- For each WITH, if the WITH level of the WITHed unit is less than the
-- WITH level of the WITHing unit, then this WITH is either unnecessary
-- or can be moved inward.  See the comments below for details.
--
procedure Check (R : Node;
                 L : Nwla.Node_With_Level_Body) is
   R_F  : constant Node_Form := Fetch (R);
   C    : Node;
   C_F  : Node_Form;
   W    : Node;
   W_F  : Node_Form;
   Wl   : Node;
   Wl_F : Node_Form;
   Wr   : Node;
   Wc   : Nwla.Node_With_Level_Body;
begin
   if Referent (Fetch (Op (R_F))) /= N_Language_Spec then
      -- First process the ancestors
      Check (Referent (Fetch (Arg (R_F, 1))), Nwla."+" (L, 1));
      -- Now look at the context clauses of this unit
      C := Arg (R_F, 2);
      C_F := Fetch (C);
      for I in 1 .. Num_Args (C_F) loop
         W := Arg (C_F, I);
         W_F := Fetch (W);
         -- Consider only the withs (ignore pragmas and uses)
         if Referent (Fetch (Op (W_F))) = N_With then
            Wl := Arg (W_F, 1);
            Wl_F := Fetch (Wl);
            for J in 1 .. Num_Args (Wl_F) loop
               Wr := Referent (Fetch (Arg (Wl_F, I)));
               Wc := Nwla.Fetch (Wr).all;
               if Nwla."=" (Wc, 0) then
                  --
                  -- If the node_with_level attribute is
                  -- zero, then this WITH is unnecessary
                  --
                  Em.Warning (Lm.Name_Of (R.U) &
                              " does not need to WITH " &
                              Lm.Name_Of (Wr.U) & ".");
               elsif Nwla."<" (Wc, L) then
                  --
```

```
                -- if the node_with_level is nonzero, but less
                -- than the level of the unit containing the
                -- with, then the WITH can be moved inward (e.g.,
                -- from a specification to a body, or from a
                -- unit to one of its subunits).
                --
                Em.Warning (Lm.Name_Of (R.U) & " WITHs " &
                        Lm.Name_Of (Wr.U) & ". " & Name_Of (L) &
                        " should WITH " & Lm.Name_Of (Wr.U) &
                        " instead.");
              end if;
            end loop;
          end if;
        end loop;
      end if;
    end Check;
  begin
    Check (Root_Of (Lm.Primary_Unit), 1);
  end Check_Unit_And_Ancestors;
begin
  Interpret_Invoking_Command;

  -- Load the specified library
  begin
    Lm.Load (Lib, Library_Name.all);
  exception
    when Lm.No_Such_Library  =>
      Em.Fatal_Error (Library_Name.all & " is not a library");
    when Lm.Cannot_Load =>
      Em.Fatal_Error ("cannot load library " & Library_Name.all);
    when Lm.Invalid_Name     =>
      Em.Fatal_Error (Library_Name.all & " is an invalid library name");
  end;

  -- Load the specified primary unit
  begin
    Lm.Load_Reserved (Lm.Primary_Unit, Lib, Unit_Name.all);
  exception
    when Lm.No_Such_Unit     =>
      Em.Fatal_Error (Unit_Name.all & " was not found");
    when Lm.Cannot_Load =>
      Em.Fatal_Error ("cannot load unit " & Unit_Name.all);
    when Lm.Invalid_Name     =>
      Em.Fatal_Error (Unit_Name.all & " is an invalid unit name");
  end;
```

```
   -- Load the standard declarations
   begin
      Lm.Load_Reserved (Lm.Language_Unit, Lib, "standard!");
   exception
      when Lm.No_Such_Unit     =>
         Em.Fatal_Error ("standard! was not found");
      when Lm.Cannot_Load =>
         Em.Fatal_Error ("cannot load unit standard!");
   end;

   -- Load all of the units in the context
   Cc.Load_Context (Errors);
   if Errors > 0 then
      Em.Fatal_Error
          (Natural'IMAGE (Errors) & " detected when loading context");
   end if;

   -- load or create any additional attributes that are needed
   Initialize_Context_Attributes;

   -- Mark all library units according to the references in the primary unit
   -- and its ancestors
   Mark_Unit_And_Ancestors;

   -- Now check the WITHs of the primary unit and its ancestors
   Check_Unit_And_Ancestors;
exception
   when Dci.Fatal_Error_Abort  =>
      null;
end With_Check;
```

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE 30 July 1990	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE STARS Intermediate Language Assessment, IRIS/DIANA Analysis	5. FUNDING NUMBERS STARS Contract F19628-88-D-0031
6. AUTHOR(S) William P. Loftus William B. Easton Frank P. Tadman	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Unisys Corporation 12010 Sunrise Valley Drive Reston, VA 22091	8. PERFORMING ORGANIZATION REPORT NUMBER GR-7670-1158(NP)
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of the Air Force Headquarters, Electronic Systems Division (AFSC) Hanscom AFB, MA 01731-5000	10. SPONSORING MONITORING AGENCY REPORT NUMBER 01430
11. SUPPLEMENTARY NOTES ([illegible])	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This technical report analyzes the requirements of the STARS environment from the perspective of intermediate language use and programmatic access to compiler information. The report details several strategies for intermediate language use in an environment for the purpose of tight integration of tool suites. In addition a detailed analysis of the currently available STARS intermediate languages is presented. These include DIANA (Descriptive Intermediate Attributed Notation for Ada), IRIS (Intermediate Representation Including Semantics), and ASIS (Ada Semantics Interface Specification). The detailed analysis compares each intermediate language with the purposes with which it was designed and the overall goals of the STARS environment. The report concludes with a recommendation to standardize on a hybrid - IRIS as the intermediate language and ASIS as a programmatic interface. It also provides a strategy to accomplish the standardization. An appendix with examples of intermediate language use is included.

14. SUBJECT TERMS Intermediate Language (IL) Descriptive Intermediate Attributed Notation for Ada (DIANA) IRIS	15. NUMBER OF PAGES 41
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR