2

SDRL Q14-02021-C

Q14 - Standards Development Plan
Ada Interfaces to DIANA
Analysis and Recommendations

Prepared for
Software Technology for Adaptable
Reliable Systems (STARS)
Unisys STARSCenter
Shipboard and Ground Systems Group
Reston, Virginia

Prepared by
Unisys Corporation
Defense Systems
System Development Group
P.O. Box 517, Paoli, PA 19301-0517

Contract No. F19628-88-D-0031
IDWO P.O. 010412

March 20, 1989

1 Executive Summary

Intermediate languages (IL) for Ada will be an important component of a robust Ada software engineering environment. The Descriptive Intermediate Attributed Notation for Ada (DIANA) is the best known and most fully explored Ada IL, and had been used as the basis of commercial Ada compilers. Unfortunately, commercially available compilers do not provide public interfaces to the Ada library, and so semantically analyzed Ada intermediate representations are not available to third party software tool developers.

Nevertheless, public tool interfaces to Ada libraries in IL form must be considered an important component of the future STARS SEE. Peregrine Systems prototype DIANA implementation, put in the public domain by STARS, is the first publicly available Ada frontend which produces a well-defined IL, based upon the 1986 definition of DIANA ~~[MS86]~~.

Given the importance of ILs to STARS, and the lack of available alternative IL implementations for STARS use, we confess to a bias to evaluate DIANA favorably as an interface formalism. Q14 expended considerable effort in testing, ruggedizing, and enhancing the Peregrine prototype in order to be able to make immediate use of it as a STARS component.

As a result of these efforts Q14 considers the current DIANA prototype to be a reasonable basis for the development of prototype tools. We recommend the Peregrine prototype be enhanced by STARS to include fuller IDL capabilities, and be retargeted to make use of more readily available parser/scanner generation tools.

Extensive use of ILs in the STARS environment raises issues concerning the integration of IL instances with other environment data; these issues must be studied. In particular, the relationship of ILs to IL *producers* (e.g., compilers, ACE) and *consumers* (e.g., test processors, browsers) must be defined and potential configuration management issues addressed.

2 Introduction

Q14 work with DIANA was intended to provide an evaluation of DIANA from two distinct perspectives. One perspective would focus on the DIANA *formalism*, i.e., does it assist or inhibit the development of software tools for Ada? In order to make such an evaluation, we needed to develop prototypical applications (application drivers) which would allow us to explore the suitability of DIANA for toolbuilding. This led to a secondary perspective which focused on the DIANA prototype *implementation*. This paper describes the Q14 efforts to evaluate DIANA from both perspectives.

Section 3 provides a brief overview of the design and rationale of the DIANA definition. Section 4 describes our evaluation of DIANA from the toolbuilder perspective. Section 5 describes our evaluation of the Peregrine prototype. Section 6 discusses some issues raised by introducing an IL processor into the STARS environment. Finally, section 7 states the conclusions made as a result of this evaluation effort.

3 History and Overview of the DIANA Definition

The DIANA Reference manual was first published in March, 1981. The initial design was based on Ada as defined in the July 1980 Ada Language Reference Manual (LRM). The design was revised to meet the July 1983 Ada LRM. The current design, dated May 1986, reflects changes made to correct weaknesses in the design and manual, and to provide a more uniform definition [MS86]. The 1986 version makes better and more extensive use of the IDL *class* mechanism to achieve a more concise and uniform specification.

DIANA is described in the Interface Description Language, IDL [JN81]. IDL is a formalism for describing *typed, attributed, directed graphs.* This formalism is obviously general enough to describe any data structure, including parse and attributed abstract syntax trees. The general purpose nature of IDL is an important factor to consider: it is not a definition *tailored* to intermediate languages, as is, e.g., IRIS [BFS88].

[MS86] makes the claim that DIANA, although designed to be an IL to provide communication between the front and back ends of a compiler, is equally suited for use by other tools in the Ada support environment. Tools such as language-oriented editors, cross-reference generators, test-case generators, formatting tools, etc., can improve the efficiency and quality of any software development process. Usually these tools use, as input and output, some intermediate form that has been produced by another tool in the environment, rather than the source text itself. Using an intermediate form precludes the need for the tool to generate lexical, syntactic, and semantic information that has usually been generated previously elsewhere in the environment. Since the generation of this information for Ada is a complex and expensive task, it makes sense to do the generation once and allow multiple tools to process the resultant structures.

While DIANA does not include information from dynamic semantic analysis, optimization, or code generation, it does provide the results of static semantic analysis, lexical, and syntactic information. This information should be sufficient to support the development of a broad class of software tools, such as those cited above. However, it was clear even to the designers of DIANA that there was a conflict between the desire to create a formalism which would support generalized tool requirements, and the need to define a concise, implementable DIANA specification.

The approach taken to resolve this conflict was to define in DIANA the *minimal* number of semantic attributes (i.e., those attributes required to represent the results of static semantic evaluation e.g., overload resolution, type checking) that could not otherwise be easily computed in one pass over the abstract syntax tree. In other words, the definition of DIANA is not sufficient to support the development of an Ada compiler – it must *a priori* be extended with additional attributes. Thus, effective use of DIANA in a CASE tool development environment presupposes the existence of a DIANA implementation which supports tool-specific extensions to DIANA.

4 Evaluating the DIANA Formalism

During the course of our evaluation, several DIANA prototype applications were developed. These prototypes were designed to facilitate evaluation of both the DIANA definition as well as the Peregrine prototype implementation. In this section we focus on the DIANA *formalism.*

DIANA is reasonably uniform, well defined, and almost intuitive. The DIANA interface package provides sufficient access to DIANA instances, and supports construction of more sophisticated access routines (e.g., generic iterators and filters) to simplify the application interface to DIANA. This approach of constructing higher-level access routines based upon the low-level DIANA interface addresses some of the issues raised below concerning the embedding of extensive navigation logic within applications.

While developing the prototypes, it also became apparent that DIANA is large and complex. Perhaps this is to be expected since Ada is a large, complex language, and it would be unnatural to assume that an Ada IL could be *simpler* than its parent language. There are two ways this complexity manifests itself to the toolbuilder: distribution of related information in DIANA instances, and the need for extensive case analysis.

4.1 Distribution of Information

One desirable property in an IL is locality of related information. That is, within a given context in an DIANA instance, the information describing that context ought to be readily available. Since we had no other IL to compare DIANA with, we are unable to claim that DIANA has this property, or that DIANA is qualitatively worse or better in this respect than other IL's. However, it is clear that quite often information which intuitively belongs to a particular DIANA construct is actually located several nodes away in the DIANA instance.

For example, it would be intuitive to assume that the name of a compilation unit would be associated with the DIANA construct which represents the compilation unit. In DIANA, however, this information requires several attribute fetches, corresponding to a mini-tree walk. Figure 1 illustrates this aspect of DIANA. Note that in this example the desired information is only two nodes away from the compilation unit node; however, the reader will appreciate that if the name of a compilation unit is two nodes distant from the compilation unit node, then in more complex situations this kind of distribution will be more exaggerated.

```
function get_compilation_unit_name(compilation_unit : tree) return string is
begin
   return printname( d(lx_symrep,
                      d(as_source_name,
                       d(as_all_decl, compilation_unit))));

end get_compilation_unit_name;
```

Figure 1: Navigational Logic in DIANA Applications.

The cumulative effect of this distribution of information is that application code can become complex, and highly dependent upon navigational logic which is really quite distinct from the application logic itself. Navigation logic in DIANA applications is a source of maintenance problems and potential run-time errors, especially when the DIANA class-structure is factored into the navigation code.

Note, however, that figure 1 encapsulates this distribution within a single utility function. Judicious use of this technique greatly enhances program understandability.

4.2 Extensive "Case" Analysis

One way in which the complexity of the DIANA definition is reduced is through use of IDL *Classes*. Classes represent an abstract node which acts as a placeholder for the class's member nodes. The class *RANGE* illustrated in figure 2 illustrates the class mechanism. In the DIANA definition, the nodes *range, range_attribute* and *void* can appear wherever the class *RANGE* is defined as the type of a structural attribute.

As noted, the 1986 definition of DIANA makes extensive use of the IDL class notation. This notation simplifies the definition considerably, especially since attributes defined in classes are *inherited* by the class members (which can also be other classes). This powerful definitional vehicle, however, results in application code which spends a great deal of time ascertaining what kind of node is currently being expected, i.e., case analysis.

```
--      RANGE ::=                range | range_attribute | void;
--      RANGE =>                 sm_type_spec : TYPE_SPEC;
--
--      range =>                 as_exp1 : EXP,
--                               as_exp2 : EXP;
--
--      range_attribute =>       as_name : NAME,
--                               as_used_name_id : used_name_id,
--                               as_exp : EXP; -- EXP or void


        procedure pp_range_class(range_class : tree) is
           subp_name : constant string := "pp_range_class";
        begin
            case kind(range_class) is
               when dn_range =>
                  pp_range(range_class);
               when dn_range_attribute =>
                  pp_range_attribute(range_class);
               when dn_void =>
                  NULL;
               when others =>
                  error(subp_name,kind(range_class));
            end case;
        end pp_range_class;
```

Figure 2: Sample code from source reconstructor.

The code in figure 2 is extracted from the source reconstructor application. The procedure **pp_range_class** processes the RANGE class, and is essentially a junction which dispatches the class instance to the appropriate routine. Of the 623 nodes defined in DIANA, 184 of them are classes. Thus this type of case analysis is done repeatedly, at the cost of run-time performance and source code size.

4.3 Summary Evaluation

DIANA is a complex formalism which requires some effort at mastery; this effort is not unreasonably extreme. Application code which makes direct use of DIANA, at least as defined by the interface package, tends to embed complex navigational logic and considerable amounts of case analysis. Judicious definition of higher-level interfaces to DIANA, perhaps application specific interfaces, is one way to reduce this overt complexity and facilitate development of more maintainable applications.

DIANA is a reasonable IL for tool processing. The author's instincts are that if DIANA is suitable for compiler applications, then other tools – by definition less complex tools – will be well served. However, as noted earlier, this will require DIANA implementations to support tool-specific tailoring of the DIANA specification.

5 Evaluating the Prototype

5.1 Prototype Overview

Peregrine developed a prototype Ada front end that produces DIANA from Ada source, as well as some tools to make use of the DIANA. This implementation is based upon the 1986 specification of DIANA which is presented in [MS86]. Access to the DIANA nodes is based on the package provided in the 1983 DIANA definition, briefly described in section 4. The prototype, which is written in Ada, produces DIANA representations from statically correct Ada source. It does not, however, guarantee failure on statically incorrect Ada programs, and so processes a superset of the Ada language.

Overall, we are significantly impressed by the quality of the implementation. The single most important metric for our evaluation was whether the implementation *works*; as our report on Q14 testing efforts illustrates, the prototype works very well. Further, the implementation is small, and imposes very little overhead on applications which use the DIANA interfaces.

5.1.1 Software Virtual Memory

As noted, the original development environment for the Peregrine prototype was restrictive, but not only in terms of compiler technology: real memory was restricted as well. Because of this, the prototype makes use of a *software virtual memory* scheme. DIANA nodes are represented as logical pointers into virtual memory implemented as a DIRECT_IO file with fixed size pages. These pointers are records containing simply the node type, page number, and location within the page. The node in virtual memory contains the type and the number of attributes in the node.

At first we were skeptical about this aspect of the prototype. It was thought that this implementation would make debugging significantly more difficult, and would have a serious impact on performance. However, we now consider this to be one of the best features of the implementation. Performance impact is not substantial, and is greatly offset by the capability to break the frontend into several distinct phases. Although a cursory examination of the code which uses the software virtual memory gives the impression of a very obtuse implementation, this impression is due mostly to issues of coding style. However, we have in fact encountered no problems in this part of the prototype.

5.1.2 Frontend Phases

The use of virtual memory allows the Peregrine prototype to be implemented as a a multi-pass system, with each pass being a separately compiled main program. The virtual memory files persist between phases, and so provides a means of communicating between phases. The organization of these phases is depicted in figure 3. A shell script, provided as part of the prototype controls the sequence of the frontend phases. A Q14 prototype has replaced the shell script with calls to CAIS-A process management; an alternative implementation would be to write ACE command procedures.

The first phase of the frontend is the parse phase. This phase parses the source and constructs an attributed, but unevaluated, abstract syntax tree. This abstract syntax tree is passed, via the virtual memory file, to the library phase.

In the library phase the transitive closure of all with-ed units is determined, and these units are read into virtual memory from the DIANA library. This is another benefit of the software virtual memory scheme: the Ada library is simply a collection of virtual memory snapshots, one snapshot per compilation unit. The library phase simply opens these files and relocates them into the global virtual memory file.

The semantic phase then does the static semantic analysis for the unit and the new DIANA instance is written to the DIANA library by the write library phase. Static semantic analysis essentially involves, for each Ada object reference and subprogram call, the location of the defining occurrence of the object or subprogram. The defining occurrence is represented as a tree-valued semantic attribute, i.e., a pointer to the node which defines the object or subprogram. Also, minor tree transformations are performed which replace temporary nodes built during parse phase with the proper nodes. For example, the parse phase can not distinguish between array slices and function calls. The 1983 DIANA definition specified the *apply* node be used in these contexts; the Peregrine prototype always constructs *function_call* nodes, and replaces them as needed during semantic analysis. The *apply* mechanism is superior, as we found several instances where the function node was not replaced; we believe it is better to distinguish all nodes which may require tree transformation.

Errors that may have occurred during the first three phases are reported during the error phase and if so, the write library phase is not performed.

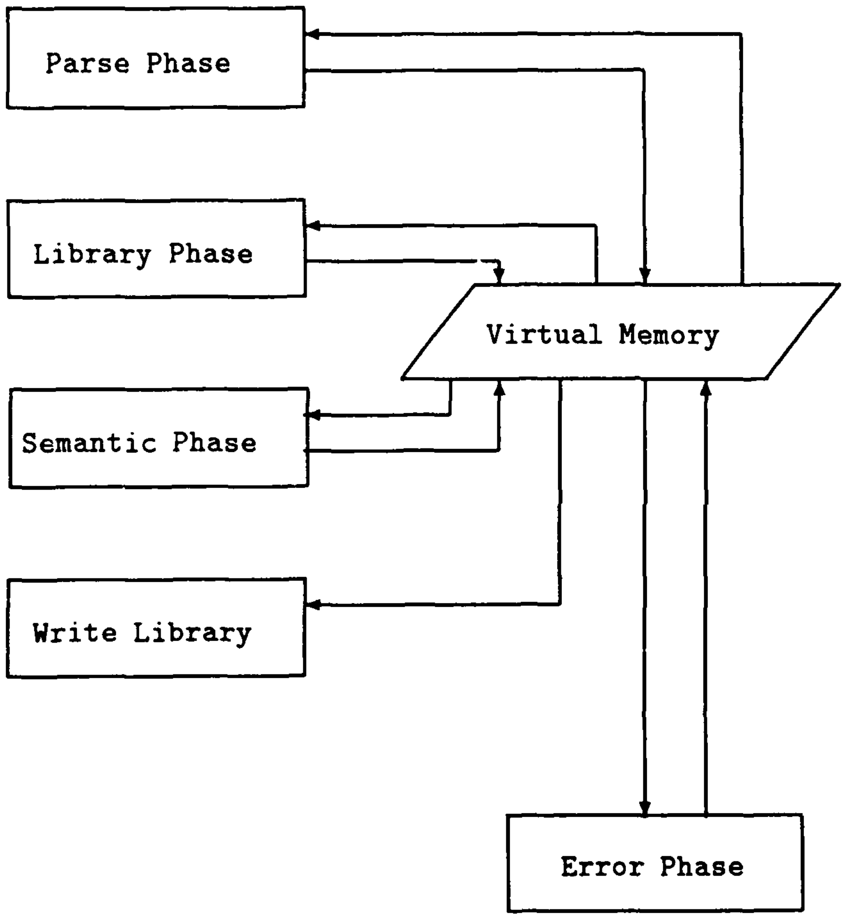

Figure 3: The five phases of the prototype Ada front end.

5.1.3 Generation Tools

DIANA was designed to have a minimum set of attributes. The attributes that make up this set were considered to be needed by most tools that might use DIANA. Rather than burden all potential users of DIANA with extra attributes, the definition of DIANA was meant to be extensible. The intent was that attributes could be added to the DIANA definition, producing a customized DIANA superset for a particular tool or environment.

IDL has the notion of IDL *refinement*, a mechanism whereby IDL *readers* (i.e. programs which read IDL) can augment or filter various attributes defined on the IDL instances. Such a mechanism would go a long way to making DIANA genuinely useful in a CASE-building setting; unfortunately, a full IDL implementation is not included in the Peregrine prototype. The prototype does provide various generation tools used to bootstrap the prototype, including a primitive IDL processor. Q14 successfully demonstrated the use of these tools to extend the DIANA definition with a new attribute. However, unless multiple DIANA implementations are to be maintained, or the DIANA implementation manages a union of all tool-specific attributes, this kind of extensibility is too limited.

5.2 Testing the Prototype

In order to evaluate DIANA as a potential IL for STARS, we believed it necessary to develop small, prototypical applications which would explore DIANA. Of course, to do this we needed a relatively stable DIANA implementation. Our first step, therefore, was to establish whether the Peregrine prototype would support our efforts; this required some testing efforts.

One important area of concern in the standards arena is that of validation to ensure compliance with a standard. Validation in general is extremely difficult because of the complexity involved in defining a finite number of tests meant to demonstrate compliance of a large body of specified functionality. Fortunately, the ACVC tests constitute a large body of well constructed validations tests. Given the lack of available alternatives to the Peregrine prototype to provide STARS with a usable Ada IL implementation, validating the Peregrine prototype became an issue.

The Interface Standards Task DIANA evaluation effort combined prototyping and validation by developing a DIANA application which would simultaneously serve to help us explore DIANA interface issues as well as make some determination concerning the soundness of the Peregrine prototype. The latter evaluation would allow us to recommend whether the Peregrine prototype should become a component, or serve as the basis of such a component, for the STARS environment.

5.2.1 Application Driver – DIANA Source Reconstructor

We decided that a source reconstructor would be a good way to test the prototype because it would access each node of a given DIANA instance. That is, in order to reconstruct the Ada source corresponding to a DIANA instance, it is necessary to perform an ordered traversal over each node in the DIANA instance. Using the DIANA instance of a given Ada source as input, the source reconstructor outputs a source file that is *semantically equivalent* to the original source.

The source reconstructor was implemented in approximately 2800 Ada source lines and has, for each kind of DIANA node, one procedure to reconstruct that particular type of node. It proved to be very helpful in showing problem areas in the Peregrine DIANA prototype, especially in abstract syntax tree construction (i.e., errors in the parse-phase tree building, not semantic analysis). Although adequate for testing the *syntactic attributes* of DIANA instances (i.e., those corresponding to the abstract syntax tree) the source reconstructor does not examine semantic attributes within a node. Since much of what a DIANA implementation is supposed to provide is described in the semantic attributes, this would be an apparent deficiency.

The Q14 plan was to first develop the source reconstructor, and then augment the implementation to include in the reconstructed source the values of various semantic attributes. For example, following the use of an object in an Ada statement, an Ada comment would repeat the type declaration corresponding to the object. This part of the source reconstructor application was never developed. As it happens, the basic source reconstructor proved sufficient not only in terms of evaluating the DIANA interfaces, but also as a vehicle for exposing errors in the Peregrine prototype. Further, the ACVC class A and C tests were determined to be a more inclusive test suite for uncovering the same class of problems the extended source reconstructor was intended to uncover.

NOTE: All such errors have been reported and fixed by Bill Easton, the prototype developer.

The source reconstructor demonstrates the need for extensibility in the DIANA implementation. Even such a simple application as a pretty printer will be hampered by DIANA because the definition does not provide enough information to allow the exact reconstruction of the original source. In this case, it is impossible to distinguish whether a reference to a function operator (e.g., "+") occurred as an infix or prefix expression. The additional information could only be placed in the DIANA instances by modifying the scanner or parser actions of the prototype.

5.2.2 Robustness Testing

The next step in testing was gradually to apply the source reconstructor to ever more complicated Ada source programs. Since simple Ada programs uncovered no problems, it was decided to attempt to determine the limits of the prototype. To do this, the source code for two STARS Foundations projects, ACE and RLF, was processed by the Peregrine implementation. This effort exposed several errors which were not related to capacity, but rather to semantically complex Ada constructs such as overloaded generic subprograms.

Eventually, the prototype successfully processed both ACE and RLF, which consist of approximately 257,000 Ada source lines and 86,100 Ada statements. This source includes both handwritten and machine generated software, and tests both capacity and correctness of the Peregrine prototype.

5.2.3 ACVC Testing

Finally, the source files of the ACVC class A and C tests were run through the DIANA prototype and the source reconstructed using the source reconstructor. Both the original source and the reconstructed source were then compiled to determine whether the resultant objects matched. As some of the ACVC tests will produce errors when compiled, only those tests that describe valid Ada were used. All of these produced equivalent objects when compiled.

An important point to note is that the ACVC testing does not imply that the Peregrine prototype successfully *passes* the A and C class tests, but rather that overt symptoms of failure (e.g., program_error exceptions) are no longer exhibited. Once again the source reconstructor proved a valuable vehicle for testing, since several ACVC sources that were "successfully" processed by the Peregrine prototype were later found to have incorrect DIANA instances – related mostly to failure to properly replace function nodes with array slice nodes after overload resolution.

5.3 Limitations and Deficiencies

5.3.1 Non-Portable Code – Representation Clauses

Our first observation is that the Peregrine prototype makes use of Ada representation clauses for the principle data type, *Tree*. Figure 4 is extracted from the DIANA source. The use of representation clauses is not of itself objectionable, and so is not considered a deficiency; we note it merely because it has prevented successful compilation on all of the Unisys STARS Ada compilers (the Alsys compiler does not support representation clauses[1]). See [PC88] for a discussion of portability problems presented by Ada representation clauses.

```
type BYTE is range 0 .. 16#FF#;
for BYTE'SIZE use 8;

type SHORT is range -16#8000# .. 16#7FFF#;
for SHORT'SIZE use 16;

type TREE is record
   TPG: SHORT;
   TTY: BYTE;
   TLN: BYTE;
end record;
```

Figure 4: Excerpt from prototype package defining TREE.

5.3.2 Extensibility

DIANA implementations must provide a mechanism for the definition and use of tool-specific attributes. Since attributes may be conveniently computed at parse (tree-building) time as well as during post semantic analysis phases, two forms of tailorability would be desirable: implementation tailorability, and DIANA tailorability.

[1]A beta version of Alsys which does support representation clauses is being tested

Since the Peregrine prototype is not implemented using readily available parser and scanner generators, substantial implementation tailorability is problematic. This conclusion is only strengthened by the lack of encapsulation of the handwritten portions of the prototype.

Although the prototype includes the means to tailor the DIANA definition, the solution is not flexible enough to support development of a wide class of tools without creating either multiple DIANA implementations, or a cumbersome, monolithic DIANA implementation which manages the union of all tool-specific attributes. Also, although we were able to extend the implementation with an additional attribute, this process was not turnkey, and exposed some hard-wired limits. These limits were easily adjusted, but this again points to the difficulties in using this implementation beyond experimental purposes.

5.4 Summary Evaluation

We accepted the Peregrine prototype for evaluation based upon a clear understanding that the implementation was an experimental prototype. Although we have criticized the implementation on various grounds, the overall quality is impressive. As a result of the extensive testing and use of the prototype, the Ada community now has a public domain Ada front end to use in the development of prototype Ada CASE tools.

Although the testing done on the Peregrine prototype is insufficient to support the claim that the prototype constructs correct DIANA instances from correct Ada source, the ACVC validation effort demonstrated that the prototype correctly constructs the DIANA abstract syntax tree, and that overload resolution is being done correctly (for the class A and C tests).

This does not demonstrate that the Peregrine prototype has *passed* the A and C tests. First, the prototype may not detect illegal Ada programs, and does not guarantee the form of the DIANA instance constructed for illegal Ada programs; thus this aspect of the ACVC is not addressed. It may be concluded, however, that the prototype constructs proper semantic interpretations (i.e., the static semantic analysis is correct) since this aspect is explicitly explored by the ACVC tests.

Overall, the prototype seems reasonably sound, and should be considered a suitable basis for construction of prototype applications. Besides the source reconstructor application, Q14 has prototyped a graphical DIANA library browser and successfully browsed the DIANA implementation itself. Thus, the prototype has sufficient capacity to process non-trivial applications, and does not impose significant resource consumption restrictions upon applications which use the DIANA interfaces.

The utility of this prototype would be dramatically enhanced if the following features or limitations would be addressed:

- Retarget the system to use gnu flex and bison, or versions of these public domain tools which generate Ada.
- Incorporate a genuine IDL processor, with at least the IDL *refinement* capability.
- Provide better type encapsulation and internal documentation.

The point of each of these items would be to facilitate the transfer of the DIANA implementation to developers interested in extending or enhancing the prototype.

6 Intermediate Languages in the STARS SEE

This section describes some issues raised by introducing Ada IL processor(s) into the STARS SEE. This issues concern required IL processor featur?s as well as larger issues of integrating IL representations into an overall environment object management system.

6.1 IL Use Issues

Two classes of IL users exist within an environment: those that *produce* IL instances, and those that *consume* these instances. It is possible to have several IL producers in addition to many consumers; this, in turn, introduces configuration management issues.

The most important IL producer within an environment is the *compiler*, since this tool produces the production code, and in a sense is the final arbiter on what constitutes valid Ada source. Unfortunately, none of the compilers in use on the Unisys STARS effort provide public access to their IL. Because of this, a separate IL producer must be introduced which certainly does not produce the *same* IL as the environment compiler(s).

Each additional IL producer introduces configuration management problems. Consider an environment in which two compilers are in use, as well as an IL producer such as the Peregrine prototype. What is the relationship between the ILs produced by the compilers to that of the Peregrine prototype? How should the IL instances by kept "in synch" with the compiler library? This might require a higher-level interface to do what amounts to dual compilation. In this scenerio, the IL producer could prove constraining, especially if the compiler can correctly process Ada source which the IL producer can not process: what if the Ada compiler succeeds but the IL generator fails?

Such issues can only be addressed when it is established which tools in the environment are IL producers, and which tools are consumers. One issue immediately apparent is the relationship of the Ada Command Environment (ACE) to STARS ILs. Current plans for ACE place it outside of the context of ILs: it is neither a producer nor consumer, but rather a separate "command language" processor which processes a small subset of the Ada language. It is reasonable, however, to evaluate the implications of extending the ACE context to include the processing of full Ada.

If ACE were to migrate towards Ada interpretation, then serious thought would have to be given to use of ACE as an IL producer *and* consumer. One mechanism for achieving this would be to introduce an ACE IL reader/writer. The reader/writer would enable ACE to maintain it's own internal IL while supporting a more accessible IL formalism (e.g. IRIS, DIANA). Such an approach could introduce problems of subset semantics, where either ACE or the external IL producers process a subset of the other's IL.

An alternative approach would be to replace the ACE IL with a better known, more complete IL, such as DIANA or IRIS. Although this would be a more involved undertaking than the above approach, it would place ACE on a migratory path to interpretation of complete Ada, and would support the emergence of a single IL producer of interest, with target code generation via compilation seen as the tail-end of the development process. In this approach, careful scrutiny of alternative ILs from the interpreter perspective would be appropriate.

Some of the characteristics of an IL to support ACE overlap with IL features which other tools can make use of. Some of these characteristics are discussed in the following subsections; these characteristics can apply both to an IL *formalism* as well as an IL producer *implementation.*

6.1.1 Lazy Evaluation

This refers to the capability of computing values *only on demand.* The utility of this feature became apparent during implementation of the DIANA browser. This prototype has substantial start-up overhead because Peregrine prototype loads the transitive closure of all units "with-ed" by the browsed unit and ensures that semantic attributes all point to valid trees. To support adequate performance for interactive tools such as the DIANA browser, either lazy evaluation should be supported, or else a monolithic, incremental data structure should be used, i.e. a persistent data structure which models the entire library and is always fully resolved.

Although this example refers to the Peregrine *loader* implementation and not to lazy semantic evaluation, it may be appropriate for environment tools to associate semantic rules with IL nodes in order to compute values on demand; such a capability can be seen as a generalization of a method used to achieve lazy evaluation of semantic attributes.

6.1.2 Incremental Processing

This refers to the capability of modifying isolated contexts within IL instances without necessarily propagating changes across an IL library or having to write an entire IL instance. This is an important concern if tools are going to *share* tool-defined attributes within the IL instances. If a tool modifies an attribute of some IL instance, then we should require a minimal propagation of changes in order to ensure that other tools will not be out-of-date with respect to the IL instance. If a crude mechanism for incrementalness is used such as timestamps, and entire IL instances are read and written in order to modify a single attribute, then extensive recompilation may be required.

6.1.3 Incompleteness

Some tools in an environment may wish to process incomplete or erroneous specifications, for instance a library browser or intelligent coding assistant. The IL formalism should have a well-defined notion of incompleteness and the IL implementation some strategy for handling illegal input strings. This characteristic should not be interpreted as requiring that the IL producer do something sensible with *all* input strings (what should it do if it is asked to process a FORTRAN program?), but rather that it should fail but still produce an IL instance which can still be processed. For example, undefined identifiers should not produce disconnected IL trees, but rather semantic attributes which indicate the location and kind of errors detected.

6.2 Environment Integration Issues

During implementation of the DIANA browser prototype, a number of issues regarding the integration of program library data and other environment data (e.g., requirements data) emerged. In general, relationships may be desired between the contents of a program library to the contents of some database; futher, the program library objects participating in these relationships may be very fine grained. For example, the value of a numeric constant within a package declaration may be tied to a requirement or a design issue discussion, which itself represented in an object base or relational DBMS.

One solution may be to integrate the IL implementation into a common environment database (or object manager). However, such an object manager would need to support management of very fine-grained objects, and would have to have suitable performance characteristics to support very demanding response-time requirements.

If no such object manager exists or can be realistically implemented, then a set of external schema interfaces need to be defined and developed such that t[illegible] ctual schema consists of various representations ranging from objects in an object manager, data in a relational DBMS, and e.g., raw Ada files as in the Peregrine prototype. This solution, however, works to the contrary of environment extensibility.

7 Conclusions

The Interface Standards task concludes that a common intermediate language and a common generator for it will be an essential part of any Ada SEE. We are inclined to conclude that further effort be expended on extending the Peregrine prototype, and that STARS should develop tools to make use of DIANA. The DIANA definition, although cumbersome in many ways, is mature and well explored, and the Peregrine prototype makes this IL available for immediate use.

An alternative IL, such as IRIS, may provide a technically superior IL formalism to DIANA. IRIS, currently being used within the DARPA ARCADIA consortium, uses a representational formalism tailored for intermediate language use, unlike IDL which is more general-purpose. Also, IRIS incorporates a highly extensible attribute implementation. The IRIS attribute mechanism may facilitate extending the semantics of the IL to incorporate special purpose sub-languages, e.g., a requirements sublanguage.

At this time we do not have enough information to recommend IRIS or DIANA as *the* IL for the STARS environment. The Peregrine prototype is more advanced in terms of accurate static semantic analysis than any other public domain Ada frontend – in Ada or otherwise. On the other hand, IRIS may represent a better long term solution for IL-based environment integration. Since it is envisioned that many tool fragments within a STARS SEE can utilize a standard IL, an analysis of IRIS versus DIANA is an essential next step for STARS.

Any IL processor inserted in the STARS environment will need to be integrated with other environment data resources. This integration introduces configuration management problems, and will stress the information management capabilities of the prototype environment.

Acknowledgement

We wish to thank Bill Easton for his consistent support, his willingness to answer questions, and the rapid turnaround on bug reports as well as extensions to the Pergrine prototype. This support

was highly appreciated and critical to the success of this evaluation.

References

[BFS88] Deborah A. Baker, David A. Fisher, and Jonathan C. Shultis. *The Gardens of Iris.* Technical Report Arcadia-IncSys 88-03, Incremental Systems Corporation, August 1988.

[JN81] D.A. Lamb J.R Nestor, W.A. Wulf. *IDL-Interface Description Language: Formal Description.* Technical Report, Carnegie-Mellon University, August 1981.

[MS86] Kathryn L. McKinley and Carl F. Schafer. *DIANA REFERENCE MANUAL Draft Revision 4.* Intermetrics, Inc., 4733 Bethesda Ave., Bethesda, MD 20814, May 1986.

[PC88] B. Pollack and D. Campbell. The suitability of ada for communcations protocols. In *Sixth National Conference on Ada Technology*, pages 170–181, March 1988.