TASK: US18
CDRL: 03089
3 June 1992

Ada Formal Methods in the STARS Environment

Informal Technical Data

STARS-SC-03089/001/00
3 June 1992

TASK: US18
CDRL: 03089
3 June 1992

INFORMAL TECHNICAL REPORT

For The

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS
(STARS)

Ada Formal Methods in the STARS Environment

STARS-SC-03089/001/00
3 June 1992

Data Type: A005, Informal Technical Data

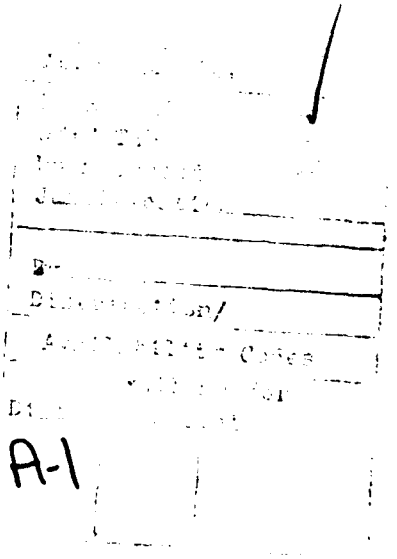
CONTRACT NO. F19628-88-D-0031
Delivery Order 003

Prepared for:

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

Prepared by:

ORA Corporation
under contract to
Paramax Systems Corporation
Tactical Systems
12010 Sunrise Valley Drive
Reston, VA 22091



TASK: US18
CDRL: 03089
3 June 1992

INFORMAL TECHNICAL REPORT
Ada Formal Methods in the STARS Environment

Principal Author(s):

D.G. Weber *Date*

Cheryl Barbasch *Date*

James Morris *Date*

Approvals:

Chief Programmer *D.G. Weber* *Date*

Thomas E. Shields 6/4/92

Task Manager *Dr. Thomas Shields* *Date*

(Signatures on File)

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	3 June 1992	Informal Technical Report

4. TITLE AND SUBTITLE

Ada Formal Methods in the STARS Environment

5. FUNDING NUMBERS

F19628-88-D-0031

6. AUTHOR(S)

ORA Corporation

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Paramax Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

8. PERFORMING ORGANIZATION REPORT NUMBER

STARS-SC-03089/001/00

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Air Force
Headquarters Electronic Systems Division
Hanscom AFB, MA 01731-5000

10. SPONSORING MONITORING AGENCY REPORT NUMBER

03089

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution "A"

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report is an interim product of Task 18. It is not intended to be a comprehensive review of work to be done on the Task, or even of work that has already been done (as the title of the deliverable might suggest). Rather, the report is a collection of several possible ways in which tools supporting formal methods might be made interoperable and/or integrated into a SEE. The possibilities discussed are merely representative, and are limited only by the effort available for completing the report. Much of the discussion relates to work being done in Ada verification at ORA, which is the STARS subcontractor primarily responsible for Task 18.

14. SUBJECT TERMS	15. NUMBER OF PAGES
SEE, formal methods, Ada, Penelope	79
	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

Contents

Chapter 1

Overview

The goal of STARS Task 18 is to develop formal methods for engineering highly dependable Ada software. The scope of this Task includes at least these areas:

- improving existing formal methods for Ada;
- implementing tools which support those methods;
- integrating the tools with each other and into a software engineering environment (SEE) for Ada.

This report is an interim product of Task 18. It is not intended to be a comprehensive review of work to be done on the Task, or even of work that has already been done (as the title of the deliverable might suggest). Rather, the report is a collection of several possible ways in which tools supporting formal methods might be made interoperable and/or integrated into a SEE. The possibilities discussed are merely representative, and are limited only by the effort available for completing the report. Much of the discussion relates to work being done in Ada verification at ORA, which is the STARS subcontractor primarily responsible for Task 18.

This chapter gives a general introduction to the subject. Chapters 2, 3 4, and 5 each discuss a solution to a specific problem in this general area.

Throughout this report, the acronym "ARM" will stand for the Ada Reference Manual [Ada83].

1.1 Formal Methods

Formal methods bring the precision of mathematical notation, theory, and reasoning to bear on the problem of understanding the behavior of computer systems. These formal methods

are intended to augment the informal methods conventionally used in system development. The conventional methods, applied to very complicated systems, have often permitted incorrect, unexpected, or undependable system behavior because of conceptual flaws in design or implementation. In contrast, formal methods lead to demonstrably correct systems, where the assumptions and reasoning on which correctness depends are made explicit.

This report deals only with a part of the area of formal methods. In general, formal methods might be applied to any phase of system development, including requirements analysis, design, implementation of both hardware and software, testing, and so forth. However, this report is restricted for the most part to formal code verification of Ada. (The single exception to this restriction is the discussion of run-time checking in Anna, in section 4.) *Formal code verification* is proof that an algorithm implemented by a piece of code meets a formally stated specification. ORA's Penelope and Ariel tools support code verification.

Formal methods may be carried out manually or with the support of automated tools. Tool support is desirable to manage the complexity of verification proofs. Unless otherwise noted, "formal methods" will be taken in this report to mean both the methods themselves and tools which support them.

1.2 Software Engineering Process

To integrate formal methods successfully into a SEE requires that one have a model of the *process* for using formal methods in conjunction with other SEE activities such as designing or debugging. Such a model will imply that tools supporting these activites have some specific relationship.

Software engineering process models that take account of formal methods are in short supply. This is primarily because formal methods (without tools) have been used on very few large software projects, and formal methods tools have been used on none. With this relatively limited base of experience, tool builders typically concentrate on supporting the process of getting verified code (which is difficult enough by itself!) rather than on how verification will affect, and be affected by, other activities.

Some attempts have been made to describe how particular formal methods should be used in practice [LST89] [Gri81]. In particular, Gries [Gri81] argues for constructing formal specifications and proofs in concert with code. Following this, ORA's Penelope verification tool provides for incremental construction of specified and verified Ada programs. This approach acknowledges that the formal specifications of a large system will probably change during the process of system development; the approach is an initial step toward a process model.

Creating a model or models of the process of using formal methods is beyond the scope of this report. However, some general observations about the process can be made.

- There will be interactions between the formal methods and other tools. As one example, a system's requirements analysis should affect the formal specifications that are

written down for the software. As another example, the properties of software that are formally specified and verified may affect code compilation and optimization. In both examples, interoperability of formal methods and conventional tools is desirable.

- Different formal methods and tools might be used complementarily in the same environment. For example, a property that is easy to prove in verification tool A might be difficult to prove, or even impossible to express, in verification tool B. If the property has meaningful consequences in B, it will be desirable for B to have access to results from A. Once again, interoperability is desirable.

Interoperability is discussed further in the next section.

1.3 Interoperability

We use the definition of this term in [DoD88]:

> *Interoperability* is defined as the ability of APSEs [Ada Programming Support Environments] to exchange database objects and their relationships in forms usable by tools and user programs without conversion.

Data and results can be exchanged between tools in many forms. In an Ada-oriented SEE, because many tools analyze Ada programs, it is convenient to standardize an intermediate representation of Ada abstract syntax to be used by all tools. In many cases the results of program analysis can be communicated from one tool to another as attributes of this abstract syntax tree. Basing interoperability on an intermediate representation saves both space and time, since it is not necessary for each tool to be able to parse and analyze static semantics of Ada: many tools can work with the intermediate representation as input. In this report, when examples of interoperability are proposed, it is assumed that the IRIS-Ada [Inc89b] [Inc89a] intermediate representation is used.

Interoperability between formal methods tools raises special concerns. Many of the current crop of formal verification tools are *formally based.* In other words, not only do these tools support reasoning about the behavior of some target program, but that reasoning is expressed in a logical framework that includes the formal semantics of the target programming language as axioms. By expressing the semantics of the target language, assumptions about the correctness of that language's compiler are made explicit. For example, the Penelope verifier allows reasoning about, and proof of, properties of Ada programs. But Penelope itself is based on a mathematical description of the semantics of Ada, and of a class of properties that can be specified for Ada programs.

Should the interoperability of formal methods tools also be formally based? If yes, then a logical framework must be found in which the semantics of different tools can be expressed, and their inputs and outputs related. Reasoning in this framework is needed to guarantee that data exchanged between two tools has the same meaning for each.

There are two views commonly expressed about this question of a formal basis for interoperability: We will call these viewpoints *relaxed* and *strict* (the terminology follows [LST89], although the discussion of them may not):

- **relaxed**: The goal of program analysis tools is to eliminate bugs and increase software dependability. Any tool that helps this goal, and any interoperability between tools that helps this goal, is desirable, even if it is not formally based.
- **strict**: A formally based verification tool, T, should use results from other tools only if those results can be related to the formal basis of T. To do otherwise undermines the confidence one can have in the results from T itself.

These viewpoints are not directly in conflict, although they do emphasize different priorities. The goal-orientedness of the relaxed viewpoint is very important and should not be neglected. However, when interoperability is discussed in this report, the strict viewpoint will be taken.

1.4 Specialized Verification Tools

Formal code verification with today's methods is very expensive. This expense comes from heavy reliance on human beings to handle time-consuming details. When tools are used to support the process, verification proofs are typically sent to an automatic theorem prover. The class of candidate theorems presented to these provers is undecidable, and so the tools cannot in principle be completely automated. In practice, finding proofs is difficult even for simple commonly occurring specifications and code fragments and so provers do not usually find verification proofs automatically. Interaction with the human programmer is essential.

Most tools accept a specification language in which a broad class of properties can be expressed about a program. This generality means that the verification methods, and theorem provers to support the verification, must be general.

Would it be an advantage to sacrifice generality in the specification language if the verification process could be made more automatic? This tradeoff could lead to cost-efficient use of verification to increase the assurance of code, even if the properties proved of the code did not completely constrain its behavior. In other words, "correctness" of a program might be underspecified, but an underspecification of "correctness" might be proved more automatically.

A tool for verifying a limited class of properties might take advantage of special algorithms or heuristics to find proofs automatically. In this way one could hope to verify simple properties of large programs using specialized verification tools, rather than what is currently the norm: verifying complicated properties of simple programs.

A familiar example of this tradeoff is found in type checking in modern programming languages: the programmer assumes that some objects, operators, and type constructors have

certain types or signatures, and then "proves" at compile time that other objects and operators defined by the program have types that will be respected at run time in every possible execution. This property might be called "type consistency". On the one hand, type consistency is a very specialized property, since it drastically underspecifies the properties a program needs to be correct. On the other hand, checking type consistency is completely automatic, and serves to reveal some conceptual errors in large programs. Ada compilers automatically verify type consistency.

Are there other classes of properties, analogous to type consistency, which most programs are expected to satisfy, and which can be verified relatively automatically? It is not necessary that the verification be completely automatic. Thus, some properties of interest which are undecidable can still be considered. However, it is necessary that an automatic analysis of property P will, for many programs occurring in practice, focus the programmer's attention on a small fraction of the code which is certain to be the source of violations of P.

The following are some classes of properties expected of most programs:

- absence of run time anomalies: in Ada terms, these anomalies include the predefined exceptions `STORAGE_ERROR` (memory exhaustion), `NUMERIC_ERROR` (arithmetic overflow and underflow), and `CONSTRAINT_ERROR` (violation of a dynamic type constraint);
- independence of a program's behavior from compiler dependences: in Ada terms, these dependences are either erroneous executions, or incorrect order dependences (ARM 1.6);
- constraints on the sequencing and timing of program actions;
- liveness, including termination and freedom from deadlock.

In most of these cases, there has been work done toward automating checks that determine when the property is violated, or conversely, checks that verify the property (e.g., [Ger81] [SI77] [FO76] [OO90] [Fre84]). Automatic checks are commonly *conservative*, in the sense that if the checks are met, the property holds, but if the checks are not met, one cannot conclude that the property is violated.

Special purpose checking or verification of properties could be integrated into a SEE for analysis of Ada. If the special classes of properties verified are expressible in the language of a more general verification tool, the tools could in principle be made to interoperate. The special purpose tools would provide specialized results that simplify the verification to be done in the more general tool.

1.5 Outline of the Report

The following chapters discuss four possibilities for making formal methods more useful in software engineering. These possibilities are just several out of many, and should not be

thought of as a complete set. Where relevant, we have discussed integration of tools into a SEE.

The first two possibilities are specialized verification tools for Ada that could be built within a SEE. Chapter 2 presents conservative checks that can be made on an Ada program which prevent some kinds of erroneous execution and incorrect order dependence in subprogram calls. Chapter 3 presents a conservative decision procedure for a class of sequencing specifications. This class is large enough to specify, among other things, that Ada program variables be given a value before they are evaluated. Interoperability between these and other tools is not discussed.

The second two possibilities involve interoperability between existing, or contemplated, formal methods tools. In chapter 4, the relation between ORA's Penelope verifier and the run-time checking tools for Annotated Ada (Anna) is investigated. The Anna language and tools have been developed at Stanford. In chapter 5, a justification for interoperability between the Ada-Ariel and AVA verification tools is worked out in great detail for a small subset of Ada. Ada-Ariel is being designed, but has not yet been implemented, at ORA. AVA is in the early stages of implementation at Computational Logic, Inc. While these two chapters do not explicitly discuss integration of the tools into a SEE, interoperability of these tools will be an important goal if they are to be used together in the same environment.

Chapter 2

Aliasing

2.1 Overview

Aliasing between two distinct occurrences of variables in a program exists if the variables represent common areas of storage. In [Gua85], two variables are defined to be potential aliases if and only if there exists a state in which they are aliases. For the following discussion, "alias" will also include "potential alias" unless specifically described as "actual alias".

In Ada, there are two categories of errors that can result from aliasing of actual parameters in a subprogram or task entry call or in a generic instantiation—erroneous execution and incorrect order dependences (see ARM 1.6). A compiler is not required to recognize these errors but if it does, it may generate the PROGRAM_ERROR exception to be raised at run time. If it doesn't, the results are undefined. If either of these two categories of errors occurs as a result of aliasing, we will say that there is an *aliasing error*. We seek conditions to be checked at compile time to guarantee the absence of these errors.

Erroneous execution can arise because the mechanism for passing parameters is not completely specified for all variable types. Array, record, and task types may be passed either by reference or by copy, as the implementation chooses. When actual parameters to a subprogram or an entry call are aliased, the results produced by the subsequent processing can depend upon the parameter passing mechanism chosen by the compiler. For example, the value of a formal parameter will be undefined whenever the actual is updated, other than by updating the formal. Any program using such an undefined value is erroneous (see ARM 6.2(13)).

Incorrect order dependence arises because the order of evaluation of subprogram or entry call parameters is not defined by the language rules (see ARM 6.4). In the case of two output parameters, if the actual parameters are aliased and the result of the subprogram call depends upon the order of copy back to the actual parameters, then there is an incorrect order dependence. For any two parameters, if the evaluation of one includes the updating of a variable that is aliased with a variable that is either read or updated by the evaluation of

the other parameter, then the order of copy in to formal input parameters or the calculation of an address for an output variable may result in an incorrect order dependence.

Calls to instantiations of generic subprograms will have the same potential aliasing errors as calls to non-generic subprograms. In addition, the evaluations of explicit generic actual parameters at the elaboration of a generic instantiation proceed in some order not defined by the language, and could thus cause an incorrect order dependence (see ARM 12.3(17)). For the execution of a task entry call, the actual parameters are evaluated as for a subprogram call, but in addition, further execution of the accept statement depends on synchronization with entry call statements (see ARM 9.5(10)).

The following sections define the specification of freedom from aliasing errors to be satisfied, present a set of conditions that are sufficient to guarantee it, and describe a tool that checks at compile time whether or not an Ada program satisfies the conditions. The tool is similar to Lint (the C program checker) in that it provides useful information to a programmer, but is conservative in its warning messages—all aliasing errors will be detected, but some correct programs may be marked as having errors.

2.2 Specification of Freedom from Alias Errors

The *visible behavior* of a program consists of its output and side-effects or changes that in some way are available to the program and can therefore affect its execution. Changes in the values of variables or values of objects pointed to by access variables are visible effects. We want to guarantee that the visible behavior of a program is not affected by aliasing of parameters.

The specification to be satisfied is the following: Neither choice of parameter passing mechanism nor order of parameter evaluation by the compiler can affect the visible behavior of the program during execution (assuming the call is not abandoned). While we will not state this specification formally, we will appeal to the Penelope formal basis ([Pol89]), the ARM, and document [Gua85] to assert that the conditions in sections 2.2.2, 2.2.3 and 2.2.4 below are sufficient to guarantee it.

The effects of I/O calls(put, get, etc.) on external files are not being considered.

2.2.1 Definitions

An *object*, a *constant* and a *variable* will be defined as in the ARM, section 3.2 and 3.2.1. Further, the following discussion assumes static semantic analysis has been done to determine the object. For example, if the name x is an actual parameter of a subprogram call, and an x is declared locally in the body of the subprogram, then they are two different objects (they have two different declarations). On the other hand, P.Q.R.x and R.x are names of the same object if R.x is a component of a record that is directly visible, with P and Q optional selectors.

Function *Lvals*(X) defines the set of objects that are modified in a given language fragment, X, and function *Rvals*(X) returns the set of objects that are read in X. For example, the expression "y + f(x)", will have Lvals(y+f(x)) consisting of all global variables modified in the body of function f, and Rvals(y+f(x)) consisting of the objects represented by y, x, and all variables read in the body of f. Constants can never be modified, so they will never be a member of Lvals(X) for any X.

For two syntactic entities, X and Y, the function *independent*(X, Y), will be defined as in section 1.6.1 of [Pol89] to be:

$$\begin{aligned} Lvals(X) \cap Rvals(Y) &= \emptyset \wedge \\ Lvals(Y) \cap Rvals(X) &= \emptyset \wedge \\ Lvals(Y) \cap Lvals(X) &= \emptyset \end{aligned}$$

Lvals(F) for function call F includes the objects modified by the body of F, and Rvals(F) includes the objects referenced by the body of F (see [Pol89], section 1.5.3).

A variable that is used as an actual parameter of a subprogram or entry call will be called *global* if it is changed or referenced in the corresponding body. The concept of global actual parameter for a generic instantiation will not be necessary, since the parameter passing mechanism is dictated by the ARM.

2.2.2 Subprogram Calls

The following two conditions are sufficient to guarantee freedom from aliasing errors for subprogram calls:

1. Actual parameters(both explicit and default) must be pairwise independent.
2. The following *no-aliasing rule* must hold.

   Aliasing of any of the three types listed below must **not** occur:

   - Aliasing between any actual parameter and a variable that is global, unless the formal parameter type is scalar or access.
   - Aliasing between two actual parameters when modes of the formal parameters are IN and OUT, or IN and IN OUT, unless either formal parameter type is scalar or access.
   - Aliasing between two actual parameters when modes of the formal parameters are both OUT, or are both IN OUT, or are OUT and IN OUT.

Independence guarantees that there is no illegal order dependence as a result of evaluation of the parameters on input to a subprogram call.

The no-aliasing rule guarantees that there is no erroneous execution and no illegal order dependence as a result of copy back to output parameters. It is the rule stated in [Gua85], with modifications for less conservative checking based on sections 6.2 and 6.4 of the ARM.

The first and second parts of the no-aliasing rule guarantee that there is no erroneous execution for variables not necessarily passed by copy. If a parameter is a scalar or an access type, then the actual is guaranteed to be passed by copy, and thus can be accessed only by its formal parameter.

The second part of the no-aliasing rule guarantees there is no erroneous execution involving non-scalars when one of the parameters is mode IN (input only). There is clearly no incorrect order of copy back.

The third part of the rule guarantees that there is no erroneous execution for non-scalars, for output modes. It is not sufficient to consider only non-scalars as in the second part of the rule since illegal order dependence can result at the time of copy back to output parameters for both scalars and non-scalars. Thus, in addition, this part of the rule guarantees that there is no incorrect order of copy back for any variable type.

Note that aliasing is allowed between two actual parameters of a subprogram call when both are mode IN.

The above conditions conservatively assume that all IN or IN OUT parameters are referenced, and that all OUT or IN OUT parameters are modified in the subprogram body. Note that there are other cases of erroneous execution or incorrect order dependence that are not a result of errors in parameter passing. These conditions only guarantee that aliasing errors that involve actual parameters will be caught.

2.2.3 Generic Instantiation

The same conditions detailed in section 2.2.2 are sufficient to prevent errors due to aliasing of actual parameters in calls to an instantiated subprogram. Note that any variable that was used as an actual parameter in the instantiation of an object of mode IN OUT will also be a global if it or its formal object is used in the generic body.

In addition, aliasing errors can be introduced during the instantiation of a generic unit. An actual parameter to a formal object of mode IN OUT or to a formal subprogram can introduce aliasing errors in the body of the generic unit being instantiated. For example, if GP is a generic subprocedure or package body, and if "Pcall(g, F);" is a call in GP where g is a formal object of mode IN OUT and F is a formal subprogram, then if the actual object matched with g is updated in the body of the actual function matched with F, then there is an aliasing error introduced by the instantiation.

The following conditions are sufficient to prevent these additional errors that occur at the time of instantiation:

- The explicit generic actual parameters must be pairwise independent.
- The conditions in section 2.2.2 must be satisfied for all subprogram calls in the body of the instantiated generic unit.

The evaluations of default expressions are excluded in checking for independence because they are guaranteed to occur in the order of the generic parameter declarations and after the evaluations of explicit actual parameters (ARM 12.3(17)). Also, the name denoting a variable associated with an IN OUT parameter is evaluated when a generic instantiation is elaborated (AI-00365/05 of the Approved Ada Language Commentaries).

Note that aliasing between two generic actual parameters is allowed because there is no difference in compiler implementations as a result of the parameter matching mechanism—the matching rules for formal objects are spelled out in ARM 12.3.1. Thus there is no erroneous execution or incorrect order dependence (there is no copy back) that can result from aliasing of the actual parameters themselves. Generic parameters to other types of formal parameters (type or subprogram) are not objects that can be aliased.

2.2.4 Programs with Tasks

The conditions in section 2.2.2 are sufficient to guarantee that errors due to aliasing of entry call parameters do not occur.

However, the set of globals used for checking global aliasing and for calculating Lvals and Rvals will include all variables that are used by any task that could be active at the point of calculation, not just the variables used in the body of the accept statement. Task attributes (T'terminated, T'callable, E'count) will not be allowed as actual parameters (they will always be considered global), and will be implicitly considered members of Lvals of an expression in which the task could be active at the point of calculation of the Lvals.

Actual parameters that are task objects behave as constants of limited type (ARM 9.2), whose initial and only values designate corresponding tasks (ARM 3.2.1). Thus a task object can never be a member of Lvals of any unit or expression, nor can it be a global that is independently updated. Objects of type access to task objects will be treated as any other access type object.

Note that shared variables between tasks are not ruled out, except when they are involved in the evaluation of actual parameters.

2.3 Examples

- A call of P(x,x) will cause an aliasing error for the following procedure, which depends upon the order of copy back of the two parameters:

```
procedure P(j: in out integer; k: out integer) is
begin
        k := j + 1;
end;
```

- Let F and H be the functions shown below, and x and y be of type access to T, visible to both F and H. If P is a procedure with two formal parameters of type T, then the call, P(F, H), causes an aliasing error.

  Because of the order of evaluation of the two parameters, the value of x.all is either the value of W or the value of y.all. This affects the visible behavior of the program and is not allowed. In terms of the specification conditions above, the parameters are not independent because Rvals(F) and Lvals(H) both include x. Note that input values to P are not undefined—they will always be W1, W2.

```
function F return T is
begin
        x.all := W;
        return W1;
end;

function H return T is
begin
        x := y;
        return W2;
end;
```

- The specification conditions are necessarily conservative in that some correct programs will be marked as erroneous. Any aliased call to the following swap procedure will be considered an aliasing error by the conditions in section 2.2.2, even though the correct result is produced in all situations:

```
procedure swap(j: in out integer; k: in out integer) is
temp: integer;
begin
        temp := j;
        j := k;
        k := temp;
end;
```

- Let F be a function with two parameters of a non-scalar type, T, and let c be a variable of type T, not visible within the body of F, and g be a global variable of type T visible within the body of F.

  The function call, F(g,c), will be marked as erroneous due to aliasing with a global, for either body below. However, inspection shows that in fact an erroneous execution error

occurs for the first function body shown, but not for the second function body. If the call were F(c,g), then neither contains an aliasing error, although the criteria would disallow both. This example shows that the sequence of actions in the subprogram body may be important in determining whether there is an actual aliasing error.

```
function F(x: T; y: T) return T is
z: T;
begin
        g := y;
        z := x;
        return z;
end;

function F(x: T; y: T) return T is
z: T := x;
begin
        g := y;
        return z;
end;
```

- Suppose A is an array of integers. A and A(j) are aliases. For the procedure call, P(A,A(j)), using the first procedure body below, an aliasing error occurs because of incorrect order of copy back, but using the second, it does not (k is by copy, X is IN only). Note that while every implementation produces the same result, it does have as a side effect that an IN parameter is changed.

  This example illustrates the importance of the mode and types of the parameters (no aliasing is allowed between two IN OUT variables, but is allowed between IN and IN OUT if one of them is scalar and they are independent).

```
procedure P(X: in out array; k: in out integer) is
begin
        k := k + 1;
end;

procedure P(X: in array; k: in out integer) is
begin
        k := k + 1;
end;
```

- The following example involving task entry parameters illustrates the erroneous execution that can result from dependence on the parameter passing mechanism involving aliasing with a global variable. When the execution of the entry call statement, T.E(X), begins, the actual parameter X is first evaluated as for a subprogram call. Further execution of the accept statement is synchronized. If the parameter passing mechanism chosen is pass-by-copy then when caller1 and caller2 terminate the value of X is false. If it is pass-by-reference then the resulting value of X is true.

```
procedure P is
        type Bool_Arr is array (1 .. 1) of Boolean;
        X: Bool_Arr := (1 => true);

        task T is
            entry E (u: in out Bool_Arr);
        end;

        task type caller;

        caller1, caller2: caller;

        task body T is
        begin
                loop
                    accept E (u: in out Bool_Arr) do
                        u := not u;
                    end E;
                end loop;
        end T;

        task body caller is
            T.E(X);
        end caller;
begin
    while not caller1'Terminated OR not caller2'Terminated loop
        null;
    end loop;
    ... (do something with X)
end;
```

- The following example involving tasks illustrates the incorrect order dependence that can result when two parameters are not independent. The value of I will be 2 if the parameters are evaluated left to right, but 1 otherwise. While the body of F does not update a variable directly, the attribute T'terminated is updated by the execution of the entry call E and corresponding accept statement. Thus T'terminated is a member of both Lvals(F) and Rvals(F).

```
procedure P is
        I: integer;

        task T is
                entry E;
        end;
```

```
        task body T is
        begin
                accept E;
        end T;

        function F return integer is
        begin
                if T'terminated then
                        return 0;
                else
                        E;
                        return 1;
                end if;
        end;

        function H(x,y: integer) return integer is
        begin
                return 2*x + y;
        end;

begin
        I := H(F,F);
end;
```

- The following example illustrates two types of errors due to global aliasing associated with a generic subprogram: when the global is used in the call to an instantiated subprogram, and when a call in the body of the generic subprogram is erroneous due to the instantiation.

```
-- the generic procedure:
generic
    type T is private;
    w: in out T;
    with function F(u: T) return T;
procedure GP(in_x: in T; out_y, out_z: out T);

procedure GP(in_x: in T; out_y, out_z: out T) is
begin
    w := in_x;
    out_y := F(w);
    out_z := in_x;
end;
```

```
-- the instantiation and call:
...
    type Bool_Arr is array(1 ..1) of boolean;
    x, y, z: Bool_Arr := (1 => true);
    function NotArr(u: Bool_Arr) return Bool_arr is
    begin
        return not u;
    end;
    function Modify_y(u: Bool_Arr) return Bool_arr is
    begin
        y := not u;
        return u;
    end;
    procedure P1 is new GP(T => Bool_Arr, w => x, F => NotArr);
    procedure P2 is new GP(T => Bool_Arr, w => y, F => Modify_y);
...
    begin
    ...
        P1(x, y, z);
        -- if x is passed by copy, z will be true,
        -- if x is passed by reference, z will be false.
        P2(x, y, z);
        -- no aliasing error caused by parameters x, y, and z, but
        -- if parm w in F(w) in GP is passed by copy, y will be true,
        -- if w is passed by reference, y will be false.
    ...
    end;
```

- When the following generic subprogram is instantiated, there is an incorrect order dependence during the evaluation of the actual explicit parameters. There is no error with the implicit actual parameter.

```
-- the generic procedure:
generic
   w: integer := 200;
   z: integer := w * w;
procedure GP(out_y: out integer);

procedure GP(out_y: out integer) is
begin
    out_y := w - z;
end;
```

```
-- the instantiation and call:
...
   X, y: integer := 0;
   function F return integer is
   begin
       X := X + 1;
       return X;
   end;
   procedure GP1 is new GP(w => F, z => F);
   procedure GP2 is new GP(w => F);
...
   begin
   ...
       GP1(y);
       -- if w before z, then  y will be -1.
       -- if z before w, then y will be 1.
       GP2(y);
       -- no aliasing error.
   ...
   end;
```

2.4 Alias-Checking Tools

Based on how conservative the checking is to be and on the purpose for which the checks are being made, several alias-checking tools may be implemented. Conservative checking for the absence of aliasing generally flags many potentially erroneous situations, even if the program is otherwise correct. Ideally it should give information with a high degree of relevance; if it is too conservative, the number of warning messages will be unacceptable if the fraction of real errors that are uncovered is small. The checks would be more useful if extra work were done to verify in certain cases of aliasing that no erroneous program results. It would be nice to verify that `swap(x,y)`, for example, is not erroneous for any input, or to verify the non-erroneous use of global variables as illustrated in example 2.3 above. Cases of extra checks that can be done to allow non-erroneous use of aliases are enumerated below, but checking for them involves determining the relative order of use of global variables and parameters, and whether they are changed or referenced. This can be done in a conservative way using the techniques developed by the definedness checking tool described in section 3.

First, checks can be made that input parameters are actually referenced and output parameters are actually modified by the body of the subprogram when considering the cases in section 2.2.2. Second, given that there is no incorrect order dependence on input to the subprogram being called, then the following additional checks can be done to allow further non-erroneous use of aliasing:

- Allow aliasing between a global and an actual IN parameter expression that is not the name of an object. If it is not the name of an object then its value is effectively passed by copy.
- Allow aliasing with a global when the actual parameter is non-scalar if the mode is IN, and the global is referenced but not updated in the called body. Referencing the global is allowed since IN parameters also can not be updated and thus there will be no undefined value.
- Allow aliasing between an actual IN parameter and a global variable that is updated in the called body if the IN formal variable is not referenced after any change to the global variable.
- Allow aliasing between an OUT or IN OUT actual parameter and global variable if the global is not changed at all and is not referenced by the subprogram after any change to the OUT or IN OUT formal variable.
- Allow aliasing between an IN variable and an OUT or IN OUT actual parameter when both are non-scalar in the following situation: if the IN formal variable and any global that may be aliased with it are not referenced in the called subprogram after any change to the OUT or IN OUT formal variable. Note that the conservative checks can not be relaxed for aliasing between an output actual parameter and a global that is changed in the called subprogram.

2.5 Implementation of an Aliasing Checking Tool

An alias-checking tool, CHECK_ALIAS, is being developed which checks the conditions in section 2.2 for all subprogram calls and generic instantiations in a given Ada compilation unit. Input is an attributed IRIS tree, resulting from running the Ada-to-IRIS tool with static semantic analysis (see [Inc89b] and [Inc89a]). The conditions given in Section 2.2 are an extension of those used by the Penelope Verification System for alias checking. The checks used here are less conservative than those done by Penelope. For its verification processing, Penelope requires annotations and assertions to be added to the Ada program. Since CHECK_ALIAS only implements the alias-checking capability of Penelope, it does not require annotations. For example, Penelope uses the fact that every global used in the subprogram is identified in an annotation, while the CHECK_ALIAS tool determines the globals used from the Ada code itself.

Output will be warning messages to the user (as in Lint) whenever any potential aliasing error discussed above is encountered. In addition, throughout all processing, results indicating an erroneous situation, or a potentially erroneous situation are stored as attributes of the appropriate IRIS node for use by other tools or for incremental analysis.

2.5.1 Definition of Aliasing

The terminology presented in sections 3.3.1 and 3.3.2 of [Gua85] and the definitions from section 2.2.1 above will be used to define *actual alias* and *potential alias* of two variables or expressions.

The name of a variable will either be an identifier, an indexed component, a slice, or a selected component. The *abstract address* of a variable in some machine state S is an ordered pair consisting of an identifier and a (possibly null) sequence of subscripts and field names called a *selector sequence*. The identifier will be the name itself, or a prefix (in the case of indexed and selected components and slices). Note that if selectors are used for visibility, then two identifiers are equal if they represent the same object. For example, P.Q.R equals R if R is a record and P and Q are optional selectors. P.Q.R and R will not be equal if P.Q.R is a different object from R and thus requires P and Q to distinguish it.

Two variables x and y with abstract addresses (I_x, $< s_x >$) and (I_y, $< s_y >$) will be (actual) aliases in state S if and only if I_x and I_y are the same identifier, and one of the selector sequences, s_x and s_y, is an initial segment of the other when both are evaluated in state S. They will be potential aliases if and only if there exists some state in which they are actual aliases.

The abstract address for all variable types is determined, for some state S, as follows:

- **Identifier**: The first coordinate of the abstract address is the identifier (or an object associated with it by the renaming declaration), and the second coordinate is the empty sequence, <>, which is an initial segment of any other sequence.
- **Records**: The abstract address of record object R is (R, <>) If R.x.y is a component of a record R.x, then the abstract address is (R, <x, y>).
- **Array**: The abstract address of array A is (A, <>) If A(i,j) is an array component, then (A, <value of i, value of j>) is the abstract address. Note that when dealing with subscripts, the values of the subscripts, and not their syntactic representations, are used. Values are computed in state S.
- **Combinations of Array and Record objects**: The first coordinate of the abstract address is the first identifier of the variable expression. The second coordinate consists of the sequence of the value of each array index and each record component in order. For example, if A(i+1).N(j) is a component of an array of records, and the record field, N, is itself an array, then the abstract address is (A, <value of i+1, N, value of j>).
- **Heap variables**: Heap variables (objects pointed to by access variables) are denoted as arrays using a new entity, T*, of a new type called a *collection* (see section 3.8 of the ARM and in section 3.8 of [Pol89]). If PTR is an access type designating objects of type T, then T* denotes a one-dimensional array whose index is of type PTR, and whose components are of type T. If x is a variable of type PTR, then the object designated by x.all is rewritten as T*(x). The abstract address can thus be represented as for

arrays, i.e. as (T*, <value of x>). If T is a record type with components L and R which are themselves type PTR, then x.L.R.L (being shorthand for x.all.L.all.R.all.L) is represented as T*(T*(T*(x).L).R).L, with abstract address, (T*, < value of x.L.R, L >). Note that x and x.all are not aliases just as i and A(i) are not aliases for array A. Furthermore, T* will be a global variable of a procedure if type PTR is declared globally to the procedure body.

2.5.2 IRIS Representation

CHECK_ALIAS walks the IRIS tree of a given unit, checking all subprogram calls and generic instantiations for aliasing errors as detailed in section 2.2.

The abstract address is represented using IRIS as follows: the first component (identifier) will be an IRIS node, generally the declaration node that is pointed to by a resolved reference node. The second component will be a list of IRIS nodes. To determine if two variables represented by their IRIS abstract addresses are aliases, nodes are compared—the identifier nodes must be equal, and one node list must be a subsequence of the other. The empty list is an initial segment of any other list. A special node (an IRIS marker_node) will be used as a wild card to match any node (e.g. the sequence <node-x, node-y> will be a subsequence of <marker_node, node-y, node-z>). If a resolved node is a rename, then the corresponding node in the abstract address will be that of the object that is renamed.

The present implementation only considers potential aliases for arrays and heap variables—it does not calculate values of variables, but instead uses the marker_node for each array index and access value. It uses the type node of the object pointed to for the collection T* used to denote heap objects. For example, if A(i,j) is an array component, then ((declaration node of A, <marker_node, marker_node>) is taken as the abstract address. If T is a record type with a component, c, and x is declared an object of type access to T, then the abstract address of x.c will be represented in the IRIS interpretation as ((declaration node of T), <marker_node, declaration node of c>). If at some future time more analysis is used to determine some of the actual values represented by marker nodes, then they can be used for less conservative checking. Otherwise, the abstract addresses for all variable types are determined in a straightforward way from the definitions described in 2.5.1. Note that when referring to an abstract address in the implementation, we will mean the conservative approximation for the actual abstract address for state S if a marker_node is used in the selector sequence.

2.5.3 Determining Lvals, Rvals

The determination of the Lvals and Rvals will be conservative in that all objects are included if they appear on some execution path, even if the path is unreachable for the given input. For example, if statement, S, is "if false then x := y; end if;" then Lvals(S) contains x, and Rvals(S) contains y although "x := y" will never be executed.

For actual parameter, X, Lvals(X) and Rvals(X) are computed as described in section 2.5.3 for expressions. The mode of the parameter is ignored. At first glance, it would seem that if the formal parameter were an output mode that Lvals(X) would always be empty, but the calculation of the address of X could also include a function call in which variables are updated.

For implementation purposes, the sets Lvals(U) and Rvals(U) for a compilation unit U will not contain every possible object that is written or read, respectively. First, they need consist only of the referenced or modified variables that are global to the unit. That is, Lvals(U) is the set of objects modified in U minus the set of objects declared in U. Second, to save space and comparison time, if two variables are aliases of each other, then only the more encompassing of the two will be considered a member of the appropriate set. For example, if R is a record with component x, and R.x and R are both read in a subprogram body, U, then only R will be stored as a member of Rvals(U). Therefore, to check for independence of two expressions the implementation determines that the Lvals of each expression does not have *aliased* variables in common with the Lvals or Rvals of the other—it is not sufficient to check that the sets intersect. Similarly, a subprogram parameter will be aliased with a global if it is aliased with an element of either Lvals(U) or Rvals(U). An entry call parameter will be aliased with a global if it is aliased with an element of Lvals(T) or Rvals(T) for any task body T that could be active at the point of call.

The following describes in more detail the determination of Lvals(I) and Rvals(I) for the given language fragment, I:

- **Expression, E**: If E is a variable, then Rvals(E) consists of the variable itself, and any variables read in calculating the location of E. Lvals(E) consists of any variables written in calculating the location of E. These variables consist of any array index (including slice components) or access variables used in representing E. For example, if E is r.m(i).C(F), where r and r.m(F) are access variables to record objects of types T1 and T2 respectively, and F is a function call, then Rvals(E) is (T2, <val of r.m(i), C, val of F >), (r, <>), (T1, <val of r, m, val of i>), (i, <>), (j, <>) plus the members of Rvals(F). Lvals(E) consists of Lvals(F).

  If E is a subprogram call or entry call, then Lvals(E) consists of the actual parameters of E where modes are OUT or IN OUT, plus the set of objects changed in the subprogram body, plus the Lvals of each actual parameter expression. Rvals(E) consists of the actual parameters of E where the modes are IN or IN OUT or OUT (access type only), plus the variables referenced in the body of E, plus the Rvals of each actual parameter expression. (Lvals and Rvals are transitive—see [Pol89], section 1.5.3).

  If E is an aggregate, then Lvals(E) consists of the union of the Lvals of any objects written in evaluating the aggregate, and Rvals(E) the union of any objects read.

  For an expression consisting of more than one variable, Lvals(E) and Rvals(E) consist of the union of the Lvals and Rvals, respectively, of its parts.

- **Block Statement, B**: A block statement consists of an optional declarative part, D, a sequence of statements and sequences of statements in optional exception handlers.

Lvals(B) and Rvals(B) consist of the union of the Lvals and Rvals, respectively, of its parts.

- Lvals(D) will consist of the union of the Lvals, and Rvals(D) the union of the Rvals of any of the following expressions that occur in the declaration:
  - Any default initialization expression, for a declared variable. The variable itself is local to the block and is not included.
  - Any expression that is part of a type declaration. e.g. type A is Array(F .. x+y) of T; — Lvals(F) will be part of Lvals(D), and Rvals(F) and x and y will be part of Rvals(D).
  - Any expression that is part of a generic instantiation—either for calculating the value of IN actual parameters, or for calculating the address of IN OUT actual parameters.
- Lvals(S) and Rvals(S) for statement S will be determined in the following way:
  - Simple Statement: Lvals(S) will contain any object not local to the block on the left-hand side of an assignment statement. All objects not local to the block in expressions from the right-hand side of an assignment statement, from a return statement, subprogram or entry call statement, or from a delay statement will be included in Lvals(S) and Rvals(S) as described above for expressions.
  - Compound Statement: Lvals(S) and Rvals(S) will consist of all variables and expressions used in an if statement, case statement, loop statement, block statement, accept statement, or select statement as described above. The loop parameter specification is local to the loop and not included.
- **Subprogram Specification**: A subprogram specification will have empty Lvals and Rvals.
- **Subprogram body**: Any expression that occurs in the parameter specification will not be part of Lvals or Rvals since any default expression is evaluated at the time of a subprogram call for determining the actual parameter. Lvals and Rvals will be calculated as for a block statement, additionally eliminating formal parameters which are local to the subprogram.
- **Package body**: Lvals and Rvals will be calculated as for a block statement.
- **Package specification**: A package specification will have Lvals and Rvals determined from the visible part and optional private part as for the declarative part of a block.
- **Task body**: Lvals and Rvals will be calculated as for a block statement. In addition, the task attributes will automatically be considered as a member of Lvals.

Chapter 3

Definedness and Other Sequencing Constraints

One common cause of erroneous program execution is variables that are used before they are given a value. Detecting this situation and preventing it before running a program would reduce errors and increase dependability.

In this section, we will describe a method for automatically detecting these erroneous executions, or verifying their absence, at *compile* time. This problem is undecidable in general; however, our method will be conservative in the sense that some executions that are not erroneous will be flagged as such, but every truly erroneous execution will be detected. The method is based on data flow analysis of the kind often used in optimizing compilers. Previously, it has been applied to the analysis of FORTRAN programs [FO76].

We note that this method has been generalized to apply to a larger class of problems in which some sequence of program actions is specified [OO90]. Similarly, most of the discussion in this section will apply to a class of sequencing constraints more general than just definedness of variables. However, for simplicity we will not treat the full generality discussed in the literature.

Our presentation differs from those cited in that it focusses on decision procedures rather than on data flow algorithms. It also specializes the method to Ada programs. Any discussion of tools will assume that IRIS-Ada is used as an intermediate representation of Ada. The terminology we use will be taken from Ada.

The presentation will not be formal. In principle, however, the method should be formally justified on the basis of a formal semantics for Ada (perhaps that of Penelope or Ariel, in which case the justification would also form the basis for interoperability).

3.1 Sequencing Specifications

It is a constraint on the sequence of actions taken by a program to require that a variable be given a value before being used. Let us consider a general class of such constraints. We will use some of the basic ideas of CSP [BHR84] to define these constraints and what it means to satisfy them.

Let a single execution of a program be characterized as a sequence of actions. We will call the actions *events* and sequences of them *traces*. For a program, P, let the set of possible events be Σ. The choice of Σ depends not only on what P is intended to do, but also on the detail in which the program is to be analyzed. For example, in analyzing the definedness of a single program variable, x, the set of events might contain only the events $\Sigma = \{A, E\}$, where A means the action "assign to x", and E means the action "evaluate x".

A simple way to characterize program P's behavior is as the set of traces that stand for possible executions. So, using program P with possible events Σ, the set of traces might be $\{\langle A, E, E, E, A, E, A, E, ...\rangle, \langle E, E, E, A, E\rangle\}$, in which case there are two possible histories, one which does not terminate and so is represented by an infinite trace, and another which produces only five events in Σ. In this scheme, finite traces can represent either termination of the program or deadlock.

Our requirement that variable x be defined before being used can now be expressed as a property of P's set of traces: every E must be preceded by some A in every trace. In the set of traces above, this property does not hold because one trace begins with an E. Note that this property would be expressed in the same way even if our analysis of P had included other events in Σ, such as assignment to variables y, z, and so on.

The following notations, taken from [OO90] but with somewhat different meanings, will form our specification language. Let $A, B, C \in \Sigma$ be sets of events. A, B, and C need not be disjoint. A *subtrace* is a subsequence of a trace.

- B specifies that some event from B occurs in every finite trace;
- $[A]B$ specifies that some event from B occurs in every finite subtrace that follows an event from A;
- $B[C]$ specifies that some event from B occurs in every subtrace that precedes an event from C;
- $[A]B[C]$ specifies that some event from B occurs in every subtrace following an event from A and preceding an event from C.

Note that none of these forms is a specification of liveness.

For the variable x discussed previously, specifying that it be defined before being used is simply $A[E]$. In addition to specifying definedness of variables, there are other examples of useful program specifications taken from this class. For example:

- specifying that every path through an Ada function either end with a `return` statement or raise an exception;
- specifying that access permissions be checked before access is granted.

Suppose the behavior of program P is represented as the set of possible traces $T \subseteq \Sigma^*$. Suppose also that P's specification is some expression S in the language just introduced. If T satisfies the condition expressed by S, then we write T *sat* S and take this to mean that P implements its specification correctly.

An even larger class of sequencing constraints is used in [OO90]. It is more general in two ways. First, a pattern of events may be specified as an arbitrary regular expression. Second, one may specify that the pattern be found on all traces, as we have done, or just on some traces. The second generalization is straightforward; the first would lead to much more complicated machinery for program analysis.

3.2 Data Flow Analysis

There is no algorithm for deciding in general whether the set of traces of an arbitrary program satisfies a sequencing specification from the class we have defined. However, there can be a decision algorithm if the set of traces meets certain constraints. We will define a collection called the *extended regular expressions*, which are straightforward modification to the usual class of regular expressions [Kle56]. Each extended regular expression will denote a set of traces. We will commonly say that this set of traces is *generated* by the expression, but sometimes will intentionally ignore the distinction between the expression and the traces it generates. We will show that there is an algorithm to decide whether the set of traces generated by an extended regular expression satisfies a sequencing specification.

The method of *data flow analysis* approximates the trace behavior of a program with a larger set of traces generated by an extended regular expression. The enlarged set of traces will be called the *data flow traces.* Suppose execution of program P produces traces in set T. Suppose also that an extended regular expression, R, can be found for the data flow traces; then $R \supseteq T$. If we can determine that R *sat* S, then it follows that T *sat* S and the program has the correct sequencing behavior.

The data flow analysis is conservative in the following sense. If R *sat* S is false, there is at least one trace in R not permitted by S. Without more analysis one does not know whether this is an incorrect trace of the program in set T, or a spurious trace introduced by enlarging T to become the set of data flow traces, R.

The set of data flow traces is typically extracted from a program in an imperative programming language (such as Ada) by following the program's control flow and ignoring all state information used to determine the flow of control at branch points. An Ada conditional, for example `if b then c else d end if;`, would have as its data flow traces the union of c's traces with d's traces, appended to each trace of b. The set of data flow traces is clearly

larger than the set of actual traces: even if b always evaluates to *true* in every program execution and so branch d is never taken, the events from branch d will still appear in some data flow trace. Branching becomes non-deterministic. By ignoring state information, one aims to convert the program to a non-deterministic finite automata (NFA), whose traces will be seen to be those of an extended regular expression.

The method is called "data flow" by analogy with data flow architectures. Program constructs which do not involve branching are the data flow operators, and program branches are the data paths along which some or all of the program's state flows to the operators for processing. The most common use of data flow analysis is in optimizing compilers, where the branch-free program constucts are called "basic blocks".

This basic method does not work for Ada programs that involve recursion or concurrency. The method we present will not work for recursive programs, since these use an (effectively) unbounded stack in determining the flow of control: a recursive program does not denote a finite automaton. Data flow traces can be gotten for a concurrent program using the CSP method of parallel composition [BHR84]: extract the data flow traces for each individual flow of control; include all synchronization events in the traces; interleave the traces in all possible ways. However, general approaches to recursion and concurrency will not be considered further. Ad hoc techniques, described later, are used in place of a more general analysis.

3.2.1 Extended Regular Expressions

The ordinary *regular expressions* are defined inductively as follows. Let R_1 and R_2 be regular expressions. Then

- ϵ is the set containing the empty trace;
- x is the set containing a single trace with a single event, also denoted by x;
- $R_1|R_2$ is the union of the sets denoted by R_1 and R_2;
- R_1R_2 is the set of all traces formed by a trace from R_1 followed by a trace from R_2;
- R_1^* is the infinite set of finite traces equal to $\epsilon|R_1|R_1R_1|R_1R_1R_1|\ldots$;

The ordinary regular expressions describe sets, possibly infinite, of finite, terminating traces.

We have extended the regular expressions in order to describe infinite, non-terminating traces as well. The *extended regular expressions* are defined inductively with all the rules listed above for regular expressions, but in addition

- ϕ is the empty set of traces;

- $R_1^{\#}$ is the set of all traces formed by a trace from R_1 followed by a trace from $R_1^{\#}$.

The empty set of traces is introduced so that it is possible to describe just the infinite traces of a program: there might be none. The $^{\#}$ constructor is introduced for traces that describe infinite looping.

Several points are worth noting if the definition of extended regular expressions is to make sense. (Equality of extended regular expressions means equality of the sets of traces they generate.)

1. Prepending the empty set yields the empty set: $\phi R = \phi$.
2. Extending an infinite trace adds no new behavior. Suppose for example that $R = F|I$, where F generates only finite traces and I only infinite ones. Then $RS = FS|I$.
3. Non-termination is irreducible for non-empty sets. For example, $\epsilon^{\#} \neq \epsilon$.

The extended regular expressions will be used to describe data flow traces. Therefore, we must show that they can characterize the traces of any non-determinstic finite automata (NFA).

A standard result from automata theory is that regular expressions characterize the *terminating* traces of an NFA. We outline the proof here. Let the NFA be represented by a finite, connected graph with labeled directed edges. The labels on the edges are the events. Given a start vertex, S, and a stop vertex, F, we must show that there is a regular expression, R_{SF}, denoting precisely those traces of events gotten by traversing the graph from S to F in the direction of the edges. One possible proof of this claim proceeds by induction on the number of edges of the graph.

- **base case:** the connected graph with zero edges has one vertex, V, and traces $R_{VV} = \epsilon$.
- **induction step:** assume that for every graph of fewer than n edges, and for every pair of vertices, S and F (not necessarily distinct), the traces starting at S and ending at F are exactly those generated by R_{SF}. An arbitrary graph, G', of n edges, is composed of a graph, G, of $n-1$ edges, and an additional edge, for which there are three possibilities:

  1. the new edge may be directed from a vertex in G to a new vertex not in G;
  2. the new edge may be directed from a new vertex not in G to a vertex in G;
  3. the new edge may be directed from one vertex in G to another.

  In each case it is straightforward to construct the traces from one node X to another Y in G' using only the constructors for regular expressions, the expressions R_{SF}, and the expression e. Thus there is a regular expression, R'_{XY}, for these traces.

When infinite traces are considered also, the problem becomes slightly more complicated, but a proof similar to the previous one is sufficient. We must show that for any graph representing an NFA, and for any start vertex S in that graph, there is an extended regular expression, E_S, for the traces gotten by traversing the graph from S to any terminal vertex, or traversing it forever. Let the set of terminal vertices be $\{T_1, T_2, \ldots T_k\}$; these can be any set of vertices in G. In particular they might be chosen to be the vertices which have no edges directed away from themselves (a trace ending on such a vertex can not be extended).

First note that $E_S = R_{ST_1}|R_{ST_2}|\ldots|R_{ST_k}|I_S$, where R_{ST_i} is the regular expression constructed previously for finite traces from S to T_i, and where I_S characterizes only the infinite traces starting at S. Therefore, it is sufficient to exhibit an extended regular expression, I_S, for the infinite traces of an arbitrary graph.

We prove that I_S exists by induction on the number of edges of the graph.

- **base case:** the connected graph with zero edges has one vertex, S, and $I_S = \phi$.
- **induction step:** let graph G' with n edges be composed of graph G with $n-1$ edges, and a new edge with label e. Assume that set of infinite traces gotten by starting at an arbitrary vertex S of G is the extended regular expression I_S. Construct an extended regular expression, I'_S for the infinite traces starting at an arbitrary vertex S of G'. There are the same three cases as before.
  1. the new edge is directed from a vertex in G to a new vertex, W, in G'. Then $I'_W = \phi$, and $I'_S = I_S$ if $S \neq W$.
  2. the new edge is directed from a new vertex, W, in G' to a vertex X in G. Then $I'_W = eI_X$, and $I'_S = I_S$ if $S \neq W$.
  3. the new edge is directed from vertex X in G to vertex Y in G. Then $I'_S = I_S|R_{SX}(eR_{YX})^*I_X|R_{SX}(eR_{YX})^{\#}$.

Therefore extended regular expressions are an adequate notation for characterizing the data flow traces, both finite and infinite.

3.2.2 Decision Algorithm

There is an algorithm for deciding whether the traces generated by an extended regular expression satisfy a sequencing specification. To show this, we will consider in turn each of the four classes of specifications defined in section 3.1.

Does an extended regular expression, R, satisfy sequencing specification B? This can be decided by appeal to the following rules according to the top-level constructor of R. A proof that the question is decidable follows from induction on the structure of R.

- ϕ *sat* B

- not ϵ *sat* B
- x *sat* B iff $x \in B$
- $R_1|R_2$ *sat* B iff R_1 *sat* B and R_2 *sat* B
- R_1R_2 *sat* B iff R_1 *sat* B or R_2 *sat* B
- not R_1^* *sat* B
- $R_1^\#$ *sat* B

Note that R_1^* fails to satisfy B because it includes the empty trace, ϵ, and this fails to satisfy B.

Before handling more complicated sequencing specifications, we must introduce a predicate, *infinite*, on extended regular expressions. $infinite(R)$ is true if every trace generated by R is infinite. This predicate can be defined inductively:

- $infinite(\phi)$
- not $infinite(\epsilon)$
- not $infinite(x)$
- $infinite(R_1|R_2)$ iff $infinite(R_1)$ and $infinite(R_2)$
- $infinite(R_1R_2)$ iff $infinite(R_1)$ or $infinite(R_2)$
- not $infinite(R_1^*)$
- $infinite(R_1^\#)$

Does extended regular expression, R, satisfy sequencing specification $[A]B$?

- ϕ *sat* $[A]B$
- ϵ *sat* $[A]B$
- x *sat* $[A]B$ iff $x \notin A$
- $R_1|R_2$ *sat* $[A]B$ iff R_1 *sat* $[A]B$ and R_2 *sat* $[A]B$
- R_1R_2 *sat* $[A]B$ iff ($infinite(R_1)$ or R_2 *sat* $[A]B$) and (R_1 *sat* $[A]B$ or R_2 *sat* B)
- R_1^* *sat* $[A]B$ iff R_1 *sat* $[A]B$
- $R_1^\#$ *sat* $[A]B$

The cases for analyzing R *sat* $B[C]$ are analogous to the previous cases with $B[C]$ replacing $[A]B$, except

- $R_1 R_2$ *sat* $B[C]$ iff R_1 *sat* $B[C]$ and (R_1 *sat* B or R_2 *sat* $B[C]$)

Does extended regular expression, R, satisfy $[A]B[C]$?

- ϕ *sat* $[A]B[C]$
- ϵ *sat* $[A]B[C]$
- x *sat* $[A]B[C]$
- $R_1|R_2$ *sat* $[A]B[C]$ iff R_1 *sat* $[A]B[C]$ and R_2 *sat* $[A]B[C]$
- $R_1 R_2$ *sat* $[A]B[C]$ iff
  R_1 *sat* $[A]B[C]$ and ($infinite(R_1)$ or R_2 *sat* $[A]B[C]$) and (R_1 *sat* $[A]B$ or R_2 *sat* $B[C]$)
- R_1^* *sat* $[A]B[C]$ iff R_1 *sat* $[A]B[C]$ and (R_1 *sat* $[A]B$ or R_1 *sat* $B[C]$)
- $R_1^\#$ *sat* $[A]B[C]$

Note that the algorithm for deciding the later classes of sequencing specification depends on the algorithm for the former classes.

The justification for each of these rules is straightforward. As an example of the reasoning involved, consider the most difficult case, the fifth rule of the last set. We will justify this rule by showing that its contrapositive holds.

- If $R_1 R_2$ *sat* $[A]B[C]$ fails, it is because there is some trace, t, of $R_1 R_2$ on which an event c from C follows an event a from A with no event from B between. This can only happen in three distinct ways.
  1. c follows a in the subtrace generated by R_1. Then R_1 *sat* $[A]B[C]$ fails to hold.
  2. c follows a in the subtrace generated by R_2, and these occur in a trace of t because some trace of R_1 is finite. Then both R_2 *sat* $[A]B[C]$ and $infinite(R_1)$ fail to hold.
  3. a occurs in a finite subtrace generated by R_1, and c follows it in a subtrace generated by R_2. Then both R_1 *sat* $[A]B$ and R_2 *sat* $B[C]$ fail.
- Conversely, if any one of the three conditions above fails to hold, then $R_1 R_2$ *sat* $[A]B[C]$ fails also.

3.3 Application to Ada

Applying the analysis of the preceding sections to an Ada program is conceptually simple:

1. identify points in the flow of control that are the events of interest;
2. specify a sequencing constraint in terms of sets of these events;
3. construct an NFA, and thus an extended regular expression, that generates the traces of events of interest in the Ada program;
4. use the decision procedures in section 3.2 to determine whether the constraint is satisfied, or might be violated.

However, there are several factors that complicate this:

1. it may not be possible to determine at compile time where in the flow of control the events of interest occur, and so to determine the data flow traces;
2. the flow of control is not always defined by the ARM;
3. modular analysis of program fragments is essential for efficiency;
4. the program may involve recursion or concurrency, and so the anaylsis is not immediately applicable.

Each of these complications will be discussed in the following sections. The problem of showing definedness of Ada variables will be used as the example throughout, and some simplifying assumptions are possible when treating this example.

3.3.1 Identifying the Events

For the general sequencing problem, the events of interest may simply correspond to locations in the program text: beginning or ending of the elaboration of a declaration or declarative part; beginning or ending of expression evaluation; entry into or exit from subprograms and rendezvous; start or finish of the execution of a statement or sequence of statements.

Unfortunately, the definition of the events of interest does not always follow the program text so neatly. In some cases, we must distinguish separate occurrences of the flow of control reaching a given point in the text – different instances of a task type or of a generic, or the distinct subprogram executions or rendezvous following different formal parameter associations. A general approach to doing this is beyond the scope of the report.

For the specific case of the definedness of Ada variables, there must be a separate sequencing specification for each Ada scalar variable *declaration* in the program: the specification constrains the program to assign a value to the variable after elaboration and before evaluation.

(Variables other than of scalar type are handled later.) During execution of the program, a particular declaration may be elaborated many times, producing different object instances, but it is not necessary to consider each such instance separately. It is sufficient to attach a sequencing specification to each declaration, rather than to each object instance that comes into being by elaborating the declaration, because the specification applies to all instances. Note the following.

- In the absence of recursion, a variable declaration within a subprogram or block may produce several variable instances in a given trace, but each instance is destroyed before the next instance is created. A single sequencing specification expresses the appropriate constraint for all instances at once.
- In the presence of recursion, a single variable declaration may correspond to several variable instances that exist simultaneously. This problem could be solved by renaming the different instances. However, it will never arise in this report: our basic decision procedure does not apply to recursion, so in section 3.3.4.1 we will approximate the traces of a recursive program by traces in which there is at most a single recurrence of each subprogram.
- A variable declaration within the body of a task type or a generic corresponds to many instances of variable declarations. The sequencing specification applies generically to all of them.

A static analysis of the program is sufficient to determine which occurrences of a simple name, x, in the program are assignments to or evaluations of a particular declaration of identifier x (either as a variable or formal parameter). Therefore, we will refer simply to the variable or formal parameter x, assuming that the static analysis of names is done.

For the declaration of scalar variable x, the set of events of interest, Σ, will include:

- the set, C, of elaborations of x without initialization or default expression;
- the set, A, of assignments to x or to a formal parameter that is certainly an alias of x;
- the set, E, of evaluations of x or of a formal parameter that is potentially an alias of x.

In terms of these sets of events, the sequencing specification for definedness of scalar variable x is then $[C]A[E]$.

There are two kinds of alias for x.

1. A formal parameter of a subprogram, entry call, or generic can be an alias for x during the execution of the corresponding unit. Using the ARM rules for formal parameter

association and parameter matching, it can be determined at compile time whether a formal parameter is, is not, or is potentially an alias for x during the execution of the unit defining the formal parameter.

2. An expression involving a composite object, such as $a(i)$ where a is an array, may refer to a scalar object. It may not always be possible to determine at compile time which scalar object is referred to, but it is at least possible to determine whether the expression is potentially the scalar variable x. This will be discussed further in the next section.

Note that an analysis at compile time may not always be able to determine precisely which expressions are aliases for x, so another conservative approximation has been made. The set A has been underestimated by considering only the aliases that are certain, while the set E has been overestimated by including potential aliases. This strengthens the sequencing specification: if the new specification is satisfied, the specification with sets A and E that are accurate would also be satisfied. Examples of this conservative approximation will be explained in the following discussion.

3.3.1.1 Variable Types

The ARM (3.2.1(18)) requires that every scalar variable be defined before being evaluated. This requirement need not be enforced either during compilation or at run-time (ARM 1.6(7)), but executions that violate it are erroneous.

The ARM does not talk explicitly about definedness of composite types. However, any erroneous execution that comes from evaluating an object of composite type that hasn't been assigned a value will result from evaluation of one of the undefined scalar subcomponents of the object. Therefore, our method can focus just on detecting undefined scalar variables. Composite objects present a problem only because they embody a collection of scalar objects, and because their existence permits expressions that are aliases for scalar objects. Objects of any type, created by an allocator, will not be handled.

The analysis of each class of non-scalar types (ARM 3.3(2)) is discussed below.

3.3.1.1.1 Access Types Access variables are initialized to `null` during elaboration. Therefore, the set C for an access variable is empty, and the sequencing specification $[C]A[E]$ is trivially satisfied. This initialization means that an access variable is always defined, and so the behavior of a program is predictable even if a subcomponent of an access variable is evaluated before being assigned to – `CONSTRAINT_ERROR` will be raised in that case.

It may be desirable to specify additionally that an access variable be initialized or assigned a value other than `null`. This is a simple modification to the set D to include elaborations of access variables which are not *explicitly* initialized.

The method we are describing is not well suited to verifying that an object created by an allocator is initialized. Usually such objects are referred to by many aliases which are difficult to analyze at compile time. Applying the method to these objects would result in few verified programs.

3.3.1.1.2 Private Types The analysis of a variable of a private type declared in package P is carried out according to the concrete type of the variable as declared in the private part of P. In other words, abstraction is ignored. This is a conservative approximation, because if the concrete representation of the variable is initialized, the variable is initialized (but the converse is not true).

3.3.1.1.3 Record Types A record is a composite of variables of other types. The declaration of a record variable is implicitly a declaration of each of the component variables. Each of the implicit component declarations is analyzed according to its type, as described in this section.

Each component of each variant of a variant record is analyzed independently. A component with a default initialization expression trivially satisfies its sequencing constraint, since the set C is empty. It should be noted that this method will guarantee that evaluation of record component $r.c$ only happens after $r.c$ has been assigned to, but it does not guarantee the absence of `CONSTRAINT_ERROR` if expression $r.c$ is evaluated in a state where r's discriminants do not have selector c.

Because each record component is selected by a simple name, it is always possible to determine which component an expression $r.c$ refers to. On the other hand, evaluating the expression r without selectors must conservatively be taken as evaluation of every potential component, regardless of the value of any discriminant, and assigning to the expression r without selectors must conservatively be taken as assignment only to those components which exist in every variant. These rules are conservative because for each scalar subcomponent of r, the set A is underestimated and the set E overestimated.

3.3.1.1.4 Array Types An array is a composite of variables of a single component type. The declaration of an array variable is implicitly a declaration of each of the component variables.

It is not practical to analyze a sequencing constraint for each component of an array, however. An expression involving an array variable, a, rarely refers to a particular array component that can be determined at compile time, e.g., $a(5)$. More common are array expressions such as $a(i)$, for which the index depends on the dynamic program state.

Instead, the method is applied once to an arbitrary component of the array variable. If this component is of scalar type, then a single sequencing constraint is analyzed. Otherwise, the arbitrary component is analyzed according to its type, as described in this section. The

analysis is conservative, in the sense that the set A contains only assignments to the whole array (an underestimate), while the set E contains evaluations of any array component (an overestimate). If the sequencing constraint is satisfied for the entire array, then it is satisfied for each component.

The method could be made less conservative by using ad hoc techniques for detecting initialization of the entire array before evaluation of any component. For example, if *exp* is an expression that can be statically determined not to evaluate any component of array a, then

```
for i in a'first .. a'last loop
    a(i) := exp(i);
end loop;
```

defines every component of a, and hence defines a, before it is used. In this example, normal exit from the loop should be treated as an event in the set A of assignments to array a. This example is the most common case; others could be found.

Verifying a sequencing constraint for an array variable means that no array component is evaluated before it is used. As in the case of record variables, note that the verification does not eliminate the possibility that `CONSTRAINT_ERROR` is raised during an attempt to evaluate or assign to a component that does not exist. Also note that a consequence of ARM 3.7.1(9) is that a composite variable, once all its components are initialized, cannot gain new uninitialized components through changing of a discriminant.

3.3.2 Flow of Control in Ada

The ARM leaves some freedom to the Ada compiler to choose the sequence of events defined by a program. The set of traces corresponding to the program is greatly expanded in these cases, because all possible orderings must be considered. Certainly this affects the analysis of sequencing properties.

3.3.2.1 Incorrect Order Dependence

For some language constructs, the ARM allows a compiler to execute different parts of the construct in some order that is not defined by the language. This can affect sequencing. For example, consider the procedure call `f(g,h));`, where g and h are parameterless functions. The ARM allows a compiler to evaluate g before h, or vice versa. If g causes assignment to global x while h evaluates x, the order of evaluation may make the difference between initialized and uninitialized use of x.

This is an example of an incorrect order dependence (ARM 1.6(9)), where the choice of order affects the visible behavior of the program. We assume that the program being analyzed

has no incorrect order dependences. Conservative static checks are often sufficient to determine this (see chapter 2, for example). Once incorrect order dependence is eliminated, any arbitrary choice of ordering by the compiler can be used for sequencing analysis.

3.3.2.2 Optimization

ARM 10.6 and 11.6 discuss permissible optimizations of Ada. We assume that only optimizations that preserve the canonical order of execution are done.

3.3.2.3 Concurrency

ARM 9.3(1) allows the execution and activation of different tasks to proceed in parallel. We have not explicitly dealt with concurrency in our model of traces. Nevertheless, an overly conservative analysis can be given for the definedness of a program variable x in the presence of tasking.

If variable x can potentially be evaluated during the execution of task T, either on some path in the task or in some path in the entry call to another task, then introduce a fictitious event representing evaluation of x at the point of T's activation. This effectively requires that x be initlaized before T is activated; further analysis of T's execution is unnecessary.

Introducing fictitious evaluations of a variable overestimates the set E for the variable and is always a conservative strengthening of the sequencing specification.

3.3.3 Incremental Analysis

It is undesirable to analyze a large program in a single pass; that would take far too long for a large program, and would be costly to update. Rather, the analysis should follow the decomposition of the program into Ada program units.

3.3.3.1 Subprograms

If the analysis of a subprogram and its caller are to be done separately, what analysis results must be stored for the subprogram in order to complete the analysis of the caller?

For any particular association of actual values to formal parameters of the subprogram, there will be some extended regular expression, R, for event sequences generated by the subprogram. In the absence of exceptions, the flow of control in a subprogram has a single entry and a single exit. Therefore, each call to the subprogram will generate only a single occurrence of the subexpression, R, in the regular expression for the entire program, Q. In applying the decision procedures of section 3.2 to Q, one will need to compute for the subprogram at most the values of $R\ sat\ [A]B[C]$, $R\ sat\ [A]B$, $R\ sat\ B[C]$, and $R\ sat\ B$. Therefore, the

sequencing problem can be solved for exception-free subprograms in isolation, and only four bits must be retained from the analysis for each sequencing specification.

In the special case of definedness of variable x, only two bits per global scalar variable and per formal scalar parameter must be retained when analyzing subprogram *foo*. This is because $R\ sat\ [C]A[E]$ and $R\ sat\ [C]A$ need not be used outside *foo*:

- if x is declared within *foo* or subprograms *foo* calls (we are neglecting recursion here), then by Ada static semantic rules x cannot appear in any unit that calls *foo*. Therefore, definedness of x can be determined just by an analysis of *foo* and subprograms it calls.
- if x is declared globally to *foo*, then it cannot also be declared within *foo*. So events from C do not occur in the execution of *foo*. Therefore $[C]A[E]$ and $[C]A$ are trivially satisfied.

3.3.3.2 Exception Handling

Every subprogram has the potential to raise exceptions, if only the predefined ones. In fact, the potential exceptional control flow paths through a program are likely to outnumber the normal ones [McH84]. An efficient decision procedure for sequencing specifications must avoid spending the bulk of its time analyzing paths that cannot be taken or that do not matter.

If an exception is raised or propagated and then handled within the subprogram, the analysis of the sequencing constraint will be carried out as described in the following paragraphs, but no additional information must be saved for future use in analyzing calling subprograms. In other words, the control flow in this case still has a single entry and a single exit from the subprogram.

The more difficult case is of a subprogram, *foo*, in which some exceptions may be raised or propagated, but are not handled. Suppose *foo* is called by *bar*. There are two subcases.

1. If *bar* has no handler for any exception raised or propagated by *foo*, then its analysis will be carried out in the manner of *foo*'s.
2. Suppose *bar* has handlers for some exceptions propagated by *foo*. The extended regular expression for *foo* can be put in the form $(F|F_1|\ldots|F_n)$, where F_i denotes just the finite traces that terminate in propagation of the ith exception from *foo* that is handled by *bar*, and F denotes all other traces, i.e., those that describe normal termination, non-termination, or which propagate other exceptions.

The extended regular expression for *bar* can then be put a form containing the subexpression $(FB|F_1H_1|\ldots|F_nH_n)$, where B denotes behavior of *bar* following normal termination of *foo*, and H_i denotes the traces of the exception handler in *bar* for the ith exception propagated by *foo*.

It can be seen from the decision procedure for sequencing specifications that analysis of *bar* will in the worst case need to analyze separately each of the $n+1$ terms of the subexpression: one for normal termination of *foo*, and one for each exception that is raised or propagated, not handled in *foo*, and handled in *bar*. Four bits must be computed for each of these cases.

Because *foo* may potentially be called from many other subprograms, each having handlers for different exceptions, and each call associating different actual parameters for formals, to analyze *foo* in advance means that four bits must be stored per unhandled exception in *foo*, per sequencing constraint. This generally amounts to quite a lot of storage and computation.

The space and time needed can easily be reduced when analyzing definedness of variables, though. First, only two bits are needed instead of four, for the reasons previously noted. Second, many cases of exceptional termination need not be analyzed at all. For example, nothing need be stored for exception X and formal parameter p if p is assigned to before X can possibly be raised: this is a consequence of the decision procedure in section 3.2.2. An efficient strategy for incremental analysis of sequencing constraints in a subprogram is probably to analyze normal termination for the subprogram and store the results, while computing and storing results for exceptional termination when needed.

3.3.4 Recursion and Concurrency

Neither recursion nor concurrency are handled by the basic method. Each may be handled by a conservative approximation.

Suppose a program is being analyzed to determine if it satisfies a sequencing specification $[C]A[E]$. A call to subprogram *foo* may be replaced by an event from E if the traces of *foo* contain no event from C. This conservatively strengthens the specification, and is the basic idea exploited in the following two sections.

3.3.4.1 Recursion

The basic method, based on NFAs, does not apply immediately to recursive programs. Subprogram calls are treated essentially by in-line expansion; recursion results from a circularity in the calling dependencies, which results in an unlimited number of in-line expansions.

When analyzing definedness of variables, recursion may be eliminated from a program by replacing the first recurrence of a subprogram call with an event from E. There are now two cases, depending on the location of the variable declaration being analyzed.

1. If the variable is global to *foo*, or is a formal parameter of *foo*, then *foo* contains no event from C. Each recurrence of *foo* may be replaced by an event from E.

2. If the variable is declared in *foo*, then eliminating recurrences of *foo* eliminates events from C (declarations of new instances of the variable). However, the analysis of these

new instances would follow exactly the same form as analysis of the instance in the original call to *foo*, so no potential violations of the sequencing specification are introduced.

A method based on context free grammars might eventually prove to be more appropriate than the one based on regular expressions that we have presented.

3.3.4.2 Concurrency

section 3.3.2.3 describes a conservative approximation to the data flow traces which requires a variable that is shared between tasks to be initialized before the activation of any task in which it is visible but not declared. This section goes further to require that a variable declared local to a task must be shown to be initialized before the task does any entry call to rendezvous with another task. This is equivalent to replacing each entry call with an event from set E. It is a conservative strengthening of the specification.

3.3.5 Other Considerations

3.3.5.1 Predefined Language Environment and Machine-Dependent Ada

We assume that it is known in advance whether the subprograms and operators predefined for Ada satsify a given sequencing specification.

In the case of definedness of variables, we assume that subprograms and packages in the predefined environment, foreign language subprograms, and machine code insertions do not use or affect any user defined variable x as a side-effect. These may affect x if it supplied as an actual parameter to subprogram or entry call, and in such cases conservative assumptions are made: IN parameters are evaluated; OUT parameters are assigned to; IN OUT parameters are evaluated before being assigned to. These are conservative assumptions about the traces in these cases.

3.4 Implementation

A tool for analyzing Ada to detect the possibility, or verify the absence, of uninitialized variables could be built within the framework of the STARS SEE, using the analysis of this chapter. We will assume that this tool would make use of the SEE's capabilities for parsing Ada text and creating from it an IRIS tree containing the results of static semantic analysis. The IRIS-Ada intermediate representation would be used as input to the tool.

Attributes of the IRIS tree would hold the results of analyzing each subprogram. The information to be stored for each subprogram is described in section 3.3.3.

Chapter 4

Penelope and Anna

In this chapter, we discuss the possibility of integrating the Penelope verifier with the Anna tools for run-time checking. Both the Penelope and Anna tools will be described briefly, in sections 4.1 and 4.2 respectively. The syntactic similarity of the specification languages processed by these tools is noted in section 4.3, and the relevance of this fact for integration is discussed. Finally, section 4.4 carries out some of the analysis that must be done before integration of these tools would be meaningful. Our conclusion is that it is unlikely that Penelope and Anna can be integrated meaningfully without some significant changes to one or other of the tools.

4.1 Penelope

The Penelope verification system is under development at ORA. It is designed for formal proof of the correctness of Ada programs. The Penelope user expresses specifications in a language, Larch/Ada, that was designed for stating conditions on program execution. For example, one can state conditions on entry to and exit (including exit by raising an Ada exception) from an Ada subprogram. Based on the user's input of specifications and Ada code, the system generates *verification conditions* (VCs), which are statements in first order logic. The proof of these statements implies that the program satisfies its specification.

Penelope's specification language belongs to the family of Larch interface languages. Larch is an algebraic specification language designed by Guttag of MIT and Horning of DEC [GHW85]. The formal, theoretical basis for verification in Penelope defines the semantics for Larch/Ada specifications, and formally justifies VC generation for a subset of Ada programs.

Penelope's user interface is an editor, in which Ada code, Larch/Ada formal specifications, and verification proofs can be edited incrementally.

4.2 Anna

Anna [L+86] is a language developed at Stanford for annotating Ada programs with formal specifications. The specifications are a class of Ada program comments, and thus every Anna program is an Ada program.

There are various tools for analyzing Anna programs. The most widely known capability of the Anna tool set is the generation of run-time checks from formal specifications. It is this capability that is of primary concern in this chapter. Run-time checking complements verification as a method for increasing the assurance of code correctness, and therefore integration of Penelope with Anna tools could be synergistic.

Anna's specifications are of two forms:

1. *virtual Ada text*, which is Ada code used directly in the construction of run-time checks, but which is not executed by the program itself;
2. *annotations*, which specify that the program's state must satisfy some conditions at given points during execution.

The conditions expressible in annotations relate values of Ada objects and objects defined in virtual Ada text. The conditions may be expressed using quantification over types definable in Ada, and using Anna functions whose properties can be defined by other annotations or by virtual Ada text.

An Anna program can be converted to an Ada program in which consistency with specifications is checked at run-time. Extra computation is done to evaluate the conditions expressed in annotations at the appropriate points during execution. When the run-time checking code finds that a condition is not satisfied, the exception `ANNA_ERROR` is raised.

4.3 Formal Specifications

While the Penelope and Anna tools have different functions, they have an important similarity: both Larch/Ada and Anna are designed primarily for specifying partial correctness of Ada programs. Even more striking, the syntax of the two languages is similar for many specification constructs. This similarity of specification languages is not an accident: the initial work on Penelope's Larch/Ada was inspired by Anna.

If there were a simple relation between the semantics of specifications in Larch/Ada and in Anna, it would be possible to integrate the Penelope and Anna tools in a meaningful way. Does every Anna annotation have a corresponding Larch/Ada annotation of the same meaning, and vice-versa?

Unfortunately, a formal answer to this question is not available, because there is no formal semantics for Anna. The meaning of Anna annotations is described informally in the Anna

reference manual. The problem of formally-based integration can still be approached in two ways, however:

1. The meaning of Anna specifications might be inferred from the Ada run-time checking code that is generated from them. The formal meaning of this code can be gotten from the Penelope formal basis.

   This is not an attractive approach, because the Anna run-time checks are really secondary, being derived from an intuitive understanding of the semantics of the annotations.

2. The formal meaning of an Anna specification might be taken to be the meaning of some "corresponding" Larch/Ada specification. For this to work, a mapping from one specification language to the other must be defined, based on our intuitive understanding of each.

In the discussion that follows, we will adopt the second approach.

A correspondence between the entire Larch/Ada and Anna specification languages is not possible, for at least two reasons:

1. Anna permits virtual Ada text as part of specifications. This Ada text defines part of the execution of Anna run-time checks. There is no corresponding feature of Larch/Ada.

2. The subset of Ada that can currently be annotated by Larch/Ada is smaller than the subset covered in Anna.

For these reasons, we will try only to define a correspondence between subsets of the specification languages. The subsets will be very limited, and in particular will not include virtual Ada text. The results of this limited attempt, however, should give some indication of the problems that would be faced in extending the correspondence to larger subsets.

We let

- an Anna annotation appearing in a sequence of Ada statements correspond to a Larch/Ada embedded assertion;

- IN, OUT, and result annotations of Anna subprograms correspond to Larch/Ada subprogram annotations of the same names.

And within an annotation:

- Anna type boolean corresponds to Larch/Ada sort boolean;

- Anna type integer corresponds to Larch/Ada sort integer, i.e., the standard infinite collection of mathematical integers;
- predefined integer operators in Anna, such as +, * and =, correspond to the Larch/Ada operators defined in the predefined integer sort.

This correspondence between specification languages can probably be extended. However, already a key problem has appeared: the Larch/Ada integers, which are the infinite set of mathematical integers, have been put into correspondence with the Anna type integer, which is treated as Ada type integer in generating run-time checks. This will make it difficult to relate proofs of correctness in Penelope to run-time checking in Anna.

4.4 Tool Integration

The Penelope and Anna tools could be meaningfully and formally integrated if a result produced in one implied something concrete about results produced using corresponding specifications in the other. In this section, several possible relations between results are considered.

Given a correspondence between specifications, such as the one of the previous section, there is a mapping from a subset of programs annotated in Larch/Ada to programs with corrsponding Anna annotations. Because we are assuming the meanings of annotations are preserved under this mapping, a program with Larch/Ada specifications in this subset is correct if and only if the Anna program resulting from the mapping is correct. (We could also have set up the inverse mapping, from Anna to Larch/Ada, but that mapping won't be needed in the following discussion.) While correctness is preserved under the mapping, it is not clear what is the relation between correctness in Penelope and the absence of `ANNA_ERROR` in run-time checking. We will use the semantics of Penelope to try to determine this relation.

First Attempt

Does a proof in Penelope that a program meets its specification imply that `ANNA_ERROR` will never be raised in the execution of the program once Anna run-time checks have been generated? Unfortunately, no.

Take for illustration the basic example of a Hoare triple, and assume that every annotation in the example is Larch/Ada that maps to some well-defined Anna annotation. Suppose the VC resulting from

```
--| P
    S;
--| Q
```

is the logical term $VC1$ in Penelope. Let `Q'` be the Anna annotation into which `Q` is mapped, and let `x1`,...,`xn` be the constituent scalar program variables in `Q'`. Then the run-time check generated by Anna for `Q'` is expressed in the following code fragment:

```
--| P
    S;
    begin
        if x1'defined and ... and xn'defined then
            Qval := Q';
        else
            Qval := true;
        end if;
        if not Qval then raise ANNA_ERROR; end if;
    exception
        when ANNA_ERROR => raise ANNA_ERROR;
        when others => raise ANNA_ERROR;
    end;
    ....
--| raise ANNA_ERROR => false;
```

Note that the Larch/Ada annotation, `--| P`, has been kept, and at the end has been added the Larch/Ada subprogram annotation that says `ANNA_ERROR` is never raised. This is done so that we can use the Larch/Ada semantics to reason about the absence of run-time checking exceptions.

Suppose the VC generated by Penelope from this annotated program is the term $VC2$. It is now possible to formulate our question precisely: for every `P`, `Q`, and `S` in our (hypothetically) well-defined subsets, does $VC1$ imply $VC2$? Suppose `P` is `true`, `S` is `null`, and `Q` is `(integer'last +1) -1 = integer'last`. Then $VC1$ is trivially true, since `Q` is an expression relating mathematical integers, and `+` is associative. But $VC2$ is not true, because the Ada expression `Q'`, `(integer'last +1) -1 = integer'last`, relates Ada integers, and may raise `CONSTRAINT_ERROR`, which in the Anna run-time checks, will raise `ANNA_ERROR`. So this run-time check does not meet these specifications, and `ANNA_ERROR` might be raised even if the code without run-time checks were correct in Penelope.

Second Attempt

The problem in the first attempt comes from the difference between mathematical integer expressions and Ada integer expressions. In Larch/Ada, these expressions are defined to yield the same value whenever program variables are defined, and in addition when `CONSTRAINT_ERROR` and `NUMERIC_ERROR` are not raised. These conditions were not met in the example.

Starting from the same Hoare triple, suppose we create the (Larch/Ada) annotated program

```
--| P
    S;
    Q';
    ....
--| raise CONSTRAINT_ERROR | NUMERIC_ERROR => false;
```

and a logical term, $VC3$, results from Penelope. Then does $VC1$ and $VC3$ imply $VC2$?

For the subsets of Larch/Ada and Anna annotations that we've been considering, the answer is yes. This is a weaker result than aimed for in our first attempt. To the programmer using an environment with both Anna and Penelope tools, a result of this kind means that there are certain proofs that can be carried out in Penelope which guarantee that the Anna run-time checks will not produce observed events in any run of the program, and so need not be generated.

Third Attempt

The problem with the second attempt is a practical one. It asks a programmer to prove a VC for a program he didn't write, and a significant amount of new code would need to be put into Penelope to produce this VC.

Here is another solution. Suppose the Anna run-time checking tools were altered to distinguish between `ANNA_ERROR`, and some new exception, call it `ANNA_EXCEPTION_ERROR`, that is propagated when `CONSTRAINT_ERROR` or `NUMERIC_ERROR` is raised in evaluating an Anna annotation. So, the Anna run-time check would appear as

```
--| P
    S;
    begin
        if x1'defined and ... and xn'defined then
            Qval := Q';
        else
            Qval := true;
        end if;
        if not Qval then raise ANNA_ERROR; end if;
    exception
        when ANNA_ERROR => raise ANNA_ERROR;
        when others => raise ANNA_EXCEPTION_ERROR;
    end;
    ....
--| raise ANNA_ERROR => false;
```

As before, the Ada block is generated from the Anna spec `--|Q'`, while the specification of

this fragment will be treated as Larch/Ada. Suppose Penelope generates $VC2'$ from this fragment. Does $VC1$ imply $VC2'$? As before, the answer is yes.

In this case, verifying a program in Penelope implies that *some* Anna exceptions will not be raised. A programmer using an integrated tool set might choose to generate run-time checks, or choose to verify, or both; but if both, he can be confident that checks that could raise `ANNA_ERROR` are redundant. Unfortunately, the Anna run-time generation tools would need to be modified to make this relationship hold.

Fourth Attempt

The first three attempts were of the form: "these Penelope VCs imply that those Anna checks won't detect an error". It is natural to investigate the converse: does the absence of detected Anna errors imply anything about the provability of VCs in Penelope?

Even for small programs a verification proof applies to very many more combinations of inputs than one is able to check by running test cases through Anna. However, we can either take this question as one of academic interest, or suppose that we have some reason to restrict our test cases to those where input values meet some constraint, `R`. Then, does the absence of `ANNA_ERROR` for exhaustive testing in `R` imply that the VC for

```
--| R and P
    S;
--| Q
```

is provable?

Unfortunately, no. Suppose `R and P` is equivalent to `not x'defined` for some integer variable `x`. Suppose also `S` is `null`, and `Q` is `x=x+1;`. Then the VC generated is never provable (assuming that Penelope is consistent) because `Q` cannot hold. But

```
--| not x'defined;
    null;
    begin
        if x'defined then
            Qval := x = x+1;
        else
            Qval := true;
        end if;
        if not Qval then raise ANNA_ERROR; end if;
    exception
        when ANNA_ERROR => raise ANNA_ERROR;
        when others => raise ANNA_ERROR;
    end;
```

```
    ....
--| raise ANNA_ERROR => false;
```

will generate a provable VC in Penelope. In other words, ANNA_ERROR will never arise in these test cases.

This discrepancy comes about because of the different way definedness is handled in run-time checks and in the Larch/Ada semantics of annotations.

4.5 Conclusion

For the limited subset of specifications we have considered, it is clear that correctness of a program in Penelope is related in some way to the absence of run-time checking errors in Anna. However, this relation is not simple. Worse, it would probably become more complicated once larger subsets of the specification languages were considered.

A simpler relation could obtained for this subset if changes were made to Penelope or Anna tools, as in examples 2 and 3. Without considering these kinds of changes further, integration of the two tools is not useful.

Chapter 5

Ada-Ariel and AVA

5.1 Introduction

This chapter discusses the possibility of achieving interoperability between the AVA and Ada-Ariel verification systems. Interoperability between verification tools seeks to gain the advantages of each without sacrificing the formal soundness of either. Interoperability is highly desirable if both the AVA and Ada-Ariel verification tools are to be integrated into the Stars Software Engineering Environment.

What do we mean by "integrating AVA and Ada-Ariel"? By this we shall mean the possibility of combining program fragments verified using AVA into an Ariel verification. We shall not address the converse possibility, because the Ariel specification language is, in general, more expressive than its AVA counterpart.

With both systems, one verifies a program by showing that the programs's meaning has the properties called for by its specification. The two systems differ in the nature of the mathematical objects that constitute the meanings of programs. In the case of (nano)AVA, the meaning of a program is essentially a function from states to states (from input state to output state); the meaning of a program in the Ariel system is the set of its possible execution traces.

In order to work out our basic ideas in some detail, we shall restrict our attention to nanoAVA. NanoAVA is primordially simple. A program consists of a single procedure. The data types are integer and Boolean. There are just assignment statements and sequences of assignment statements. Expressions consist of a single identifier or relation involving identifiers (denoting variables or named constants). Keep in mind that one must learn to walk before attempting to run.

We shall take the definition of the nanoAVA semantics to be the Boyer-Moore definition given in chapter III-2 of CLinc Technical Report 21. We shall take the specification language of the AVA system to be the language of the Boyer-Moore logic. An AVA specification is anything

one can state in Boyer-Moore logic about a program's AVA meaning.

Ada-Ariel semantics are specified using dynamic structures. In this report, we will encode Ada-Ariel dynamic structures in Caliban and state their properties in Clio's metalanguage. This means that we take the latter to be the Ada-Ariel metalanguage and that an Ada-Ariel specification is anything one can state in Clio's metalanguage about a programs's Ada-Ariel meaning.

In section 2 we describe the Boyer-Moore logic and give the semantics of nanoAVA in it. In section 3 we do the same for Caliban/Clio. In section 4 we show how to translate the Boyer-Moore representation of a program and its specification into Caliban/Clio.

5.2 NanoAVA and its Boyer-Moore Semantics

5.2.1 The Boyer-Moore Logic

The semantics of nanoAVA are expressed in the language of the Boyer-Moore logic. We will begin by giving a brief summary of the latter. Our goal is to provide the reader with a knowledge of it sufficient for understanding what follows. For a complete discussion, see [BM79, BM88].

Formulas in the logic are built up from terms. A term is either a variable symbol or the application of a function of n arguments to n terms. The syntax is LISP-like. This means that a variable or function symbol consists of a string of uppercase alphanumeric and certain other characters. An application of a function `f` to its arguments `x1,...,xn` is written `(f x_1 ... x_n)`.

The logic includes equality and Boolean operators, which can be thought of as built-in functions (in the programming language sense of *function*). The constants `(TRUE)` and `(FALSE)` are 0-ary functions. There is a conditional operator, written `(IF X Y Z)`, which returns `Y`, if `X` is `(TRUE)`, and `Z`, otherwise. Equality is written `(EQUAL X Y)` and returns `(TRUE)`, if `X` and `Y` are equal. The usual propositional connectives, `AND`, `OR`, `NOT`, and `IMPLIES`, are available and have the expected meaning.

The logic provides a means for adding new "data types". This facility is provided by the *shell principle*, which is essentially a means for axiomatizing classes of inductively constructed objects. The axioms define base and constructor functions for building objects of the new type, destructor functions for taking them apart, and recognizer functions for identifying them as instances of the type. We omit the details, but note that the shell principle is used in the primitive logic, i.e., the logic before it has been extended by a user, to axiomatize the natural numbers, ordered pairs, literal atoms, and the negative integers.

Conventional data structures such as lists, tables, binary trees, etc., are constructed out of ordered pairs. The nanoAVA semantics to be presented below makes extensive use of lists, so we need to give more of the notation for functions on lists. Ordered pairs are con-

structed by the function CONS: (CONS X Y) returns the ordered pair whose first component is X and whose second component is Y. (LISTP X) returns (TRUE) (henceforth abbreviated T), if X is an ordered pair constructed by CONS and (FALSE), otherwise. (CAR X) returns the first component of X, if X is an ordered pair and (ZERO) (the base object of the natural number shell, henceforth abbreviated 0), otherwise. (CDR X) returns the second component of X, if X is an ordered pair and 0, otherwise. The literal atom NIL will be used to represent the empty list. Given a list s, the ordered pair (CONS x s) is the list obtained by adding x to the front of s. (LIST X1 ... Xn) is an abbreviation for the list (CONS X1 (CONS ... (CONS Xn NIL) ...)). Nests of CARs and CDRs can be abbreviated with function symbols of the form C...A...D...R, e.g. (CADAR X) abbreviates (CAR (CDR (CAR X))).

The QUOTE notation is provided for writing down list and atomic constants. The symbol QUOTE is used as if it denoted a function of one argument. However, QUOTE does not denote a function and its "argument" need not be a term. Rather, the entire QUOTE expression, i.e. all the symbols between the open parenthesis immediately to the left of the QUOTE and the corresponding closed parenthesis, is an abbreviation for a term. (QUOTE s), where s is a symbol or a well-balanced parenthesized string, is further abbreviated 's.

Functions are defined by equations that reduce calls of new functions to combinations of calls of previously defined ones. The form is as follows

```
(fn x1 ... xn) = term
```

There are restrictions on such equations which insure the uniqueness of the function so defined. We omit the details. The definition of recursive functions is allowed, but one is obliged to prove termination by showing the existence of an ordinal measure of the arguments of the function under definition that decreases in each recursive call.

A number of useful functions on lists and natural numbers are built into the basic theory. Examples of the former are APPEND and MEMBER. Examples are of the latter are the ordering relations and arithmetic operations.

There is an interpreter, expressed as a function in the logic, capable of determining the value of terms in the logic. This makes the logic "executable".

The logic lacks:

- quantification (unbounded, that is, except universal quantification over the entire formula)
- mutual recursion (except for the trick mentioned in [BM88] which involves defining two mutually recursive functions by means of a third function which calls one of the two depending on an argument),
- infinite objects,

- sets,
- abstraction,
- higher-order variables.

There is only a very constructive sense in which partial functions can be defined.

5.2.2 A Boyer-Moore Semantics for NanoAVA

A nanoAVA program consists of a single procedure definition. There are two data types: INTEGER and BOOLEAN. The only primitive command is assignment; the only composite command is a sequence of commands. Expressions consist of a single identifier or two identifiers and a relational operator.

Next we will give the abstract syntax of nanoAVA as represented in the Boyer-Moore logic. First we enumerate the categories of the abstract syntax and the variables that range over them.

$c \in$ Cmp compilations
$p \in$ *Sub* subprogram bodies
bdi $\in$ Bdi *basic declarative items*
$ps \in$ Ps parameter specifications
$s \in$ Stm statements
$tm \in$ Tm type marks
$e \in$ Exp expressions
$O \in$ Opr relational operators
$i \in$ Ide identifiers

Now, for each defining production, we will give that production and the corresponding Boyer-Moore term. First, we say a bit about about how the abstract syntax tree is represented. Its nodes are "record-like" structures — actually, lists whose first element is the name of the "record" and whose remaining elements contain the values of the record's fields. A predicate is defined that checks whether a list is an instance of a record by looking at that list's first element. The field names become functions which extract the values of a record's components. Such structures are defined by means of the following macro which is explained by a comment taken from CLInc Technical Report 21. Note that `defn` is a command to the Boyer-Moore theorem prover to add a function definition as a new axiom (after checking its admissibility).

```
;;; macros for declaring record-like structures
;;;
;;; (defrecord foo (bar baz ...)) expands to
```

```
;;;   (defn mk-foo (bar baz ...) (list 'foo bar baz ...))
;;;   (defn foo-p (x) (and (listp x) (equal (car x) 'foo)))
;;;   (defn foo-bar (x) (car (cdr x))
;;;   (defn foo-baz (x) (car (cdr (cdr x))))
;;;   ...
```

Instances of such structures are created by `mk-foo`, `foo-p` recognizes them, and `foo-bar`, `foo-baz`, etc. extract their components.

In the following, S^* denotes the set of sequences of elements of S and if s is a variable ranging over S, s^* is implicitly defined to be a variable ranging over S^*.

There is no need for a definition of the production $c ::= p$. To the production $p ::=$ **proc** i ps^t bdi^* s^*, corresponds the definition

```
(defrecord sub (ide ps-list bdi-list stm-list))
```

To the production $ps ::=$ **inout** i^* tm, corresponds the definition

```
(defrecord ps (ide-list tm))
```

To the production $bdi ::=$ **var** i^* tm e, corresponds the definition

```
(defrecord bdi-var (ide-list tm exp))
```

To the production $bdi ::=$ **const** i^* tm e, corresponds the definition

```
(defrecord bdi-const (ide-list tm exp))
```

To the production $s ::=$ **null**, corresponds the definition

```
(defrecord stm-null ())
```

To the production $s ::= i := e$, corresponds the definition

```
(defrecord stm-asg (var exp))
```

To the production $e ::= i$, corresponds the definition

```
(defrecord exp-ide (ide))
```

To the production $e ::= i\ O\ i$, corresponds the definition

```
(defrecord exp-rel (lhs opr rhs))
```

To the production $O ::= =| \ / =|<|<=|>|>=$, correspond the definitions

```
(defrecord opr-eq ())
(defrecord opr-ne ())
(defrecord opr-lt ())
(defrecord opr-le ())
(defrecord opr-gt ())
(defrecord opr-ge ())
```

No definition is needed for the production $tm ::= i$.

Next, we give the Boyer-Moore definition of the meaning functions for nanoAVA's dynamic semantics. The domain of values is the union of Bool and Integer. States are functions from identifiers into values.

The function that assigns meanings to compilation units has the following mathematical definition

$$E_c(\textbf{proc i } ps^*\ bdi^*\ s^*, \sigma) = E_{s^*}(s^*, E_{bdi^*}(bdi^*, \sigma))$$

In words: the meaning of a function is the meaning of its body in the state obtained from the elaboration of its local declarations in the initial state in which the function is called. The Boyer-Moore term for this equation is the following.

```
(defn e-c (x store)
  (e-s-list (sub-stm-list x)
            (e-bdi-list (sub-bdi-list x) store)))
```

The function that assigns meanings to lists of basic declarative items has the following mathematical definition

$$E_{bdi^*}(bdi^*, \sigma) = E_{nbdi^*}(N_{bdi^*}(bdi^*), \sigma)$$

The function N_{bdi^*} "normalizes" basic declarative items, i.e. it transforms items of the form **var** i^* tm e to a list of items **var** i_1 tm e, ..., **var** i_n tm e. This function has the following Boyer-Moore definition

```
(defn e-bdi-list (x store)
  (e-nbdi-list (nbdi-list x) store))
```

Note that **nbdi-list** is the Boyer-Moore counterpart of the mathematical function N_{bdi^*}.

The function that assigns meanings to lists of normalized basic declarative items has the following mathematical definition

$$E_{nbdi^*}(\Lambda, \sigma) = \sigma$$
$$E_{nbdi^*}(nbdi; nbdi^*, \sigma) = E_{nbdi^*}(nbdi^*, E_{nbdi}(nbdi, \sigma))$$

and their Boyer-Moore counterpart

```
(defn e-nbdi-list (x store)
  (if (listp x)
      (e-nbdi-list (cdr x)
                   (e-nbdi (car x) store))
      store))
```

The functions that assign meanings to normalized basic declarative items are the following

$$E_{nbdi}(\textbf{var } i \ tm \ e, \sigma) =$$
$$E_{nbdi}(\textbf{const } i \ tm \ e, \sigma) = \sigma[i \leftarrow E_e(e, \sigma)]$$

The notation $\sigma[i \leftarrow E_e(e, \sigma)]$ denotes the state that is like σ except at i where its value is $E_e(e, \sigma)$. The corresponding Boyer-Moore definition is

```
(defn e-nbdi (x store)
  (if (nbdi-const-p x)
      (update-function store
                       (nbdi-const-ide x)
                       (e-e (nbdi-const-exp x) store))
      (update-function store
                       (nbdi-var-ide x)
                       (e-e (nbdi-var-exp x) store))))
```

The mathematical function that assigns meanings to statement lists is as follows

$$E_{s^*}(\Lambda, \sigma) = \sigma$$
$$E_{s^*}(s; s^*, \sigma) = E_{s^*}(s^*, E_s(s, \sigma))$$

with its Boyer-Moore counterpart

```
(defn e-s-list (x store)
  (if (listp x)
      (e-s-list (cdr x)
                (e-s (car x) store))
      store))
```

The mathematical function that assigns meanings to statements is as follows

$$E_s(\textbf{null}, \sigma) = \sigma$$
$$E_s(i := e, \sigma) = \sigma[i \leftarrow E_e(e, \sigma)]$$

with its Boyer-Moore counterpart

```
(defn e-s (x store)
  (if (stm-null-p x)
      store
      (update-function store
                       (stm-asg-var x)
                       (e-e (stm-asg-exp x) store))))
```

The mathematical function that assigns meanings to expressions has the following definition

$$E_e(i, tm, \sigma) = \sigma(i)$$
$$E_e(i\ O\ i', tm, \sigma) = E_o(O, E_e(i, \sigma), E_e(i', \sigma))$$

with its Boyer-Moore counterpart

```
(defn e-e (x store)
  (if (exp-rel-p x)
      (e-opr (exp-rel-opr x)
             (e-e (exp-rel-lhs x) store)
             (e-e (exp-rel-rhs x) store))
      (apply-function store (exp-ide-ide x))))
```

The mathematical function that assigns meanings to the relational operators has the following definition

$$\begin{aligned}
E_O(=, v, v') &= \textbf{if } v = v' \textbf{ then } \textit{true} \textbf{ else } \textit{false} \\
E_O(/=, v, v') &= \textbf{if } v \neq v' \textbf{ then } \textit{true} \textbf{ else } \textit{false} \\
E_O(<, v, v') &= \textbf{if } v < v' \textbf{ then } \textit{true} \textbf{ else } \textit{false} \\
E_O(<=, v, v') &= \textbf{if } v \leq v' \textbf{ then } \textit{true} \textbf{ else } \textit{false} \\
E_O(>, v, v') &= \textbf{if } v > v' \textbf{ then } \textit{true} \textbf{ else } \textit{false} \\
E_O(>=, v, v') &= \textbf{if } v \geq v' \textbf{ then } \textit{true} \textbf{ else } \textit{false}
\end{aligned}$$

with its Boyer-Moore counterpart

```
(defn e-opr (opr v1 v2)
  (if (opr-eq-p opr) (if (equal v1 v2) 'true 'false)
  (if (opr-ne-p opr) (if (not (equal v1 v2)) 'true 'false)
```

```
(if (opr-lt-p opr) (if (or (and (equal v1 'false) (equal v2 'true))
                           (lessp v1 v2))
                       'true
                       'false)
(if (opr-le-p opr) (if (or (or (equal v1 false) (equal v2 'true))
                           (equal v1 v2)
                           (lessp v1 v2))
                       'true
                       'false)
(if (opr-gt-p opr) (if (or (and (equal v1 'true) (equal v2 'false))
                           (lessp v2 v1))
                       'true
                       'false)
(if (opr-ge-p opr) (if (or (or (equal v1 'true) (equal v2 'false))
                           (equal v1 v2)
                           (lessp v1 v2))
                       'true
                       'false)
0)))))))
```

5.3 An Ariel Semantics for nanoAVA

Since we will give a Clio/Caliban encoding of an Ariel semantics for nanoAVA, we begin by briefly describing Caliban and the Clio metalanguage and prover. Again, our goal is to provide the reader with sufficient knowledge to understand what follows.

5.3.1 Caliban Types

All Caliban types are complete lattices. For the purpose of this discussion this means that every Caliban type is equipped with an element, denoted $\perp$ and pronounced "bottom", which is the value of any undefined term of the type in question. Roughly speaking, a term is undefined if it is impossible to compute. In the case of functions a related notion of *strictness* is important. A function f is said to be strict, if $f\ \perp = \perp$.

Caliban provides certain built-in types. The basic types include two "sorts" of number, booleans, and characters. Characters are described according to a syntax similar to C, which we won't repeat here. The booleans are the familiar truth values plus $\perp$. The usual operations on booleans are available. The first sort of number, called *num*, is essentially the C integer type: there is no overflow and the familiar C operators are available. The second sort of number is a constructed type `NAT` of natural numbers. We will say more about `NAT` in a moment.

There are two built-in type constructors for building lists and tuples, respectively. Lists are

constructed using the binary infix operator ':' (pronounced *cons*). ':' is familiar; it makes a list whose head is its first argument and whose tail is its second argument. Lists may be given explicitly, e.g.:

$$L = [1, 2, 3, 4, 5].$$

The empty list is denoted [ ], so L can be written alternatively as

$$L = 1 : 2 : 3 : 4 : 5 : [\,].$$

The other built-in construction is the n-tuple, written

$$\ll x_0, x_1, \ldots, x_n \gg,$$

where the x_i are any legal Caliban expressions.

The type of a function is denoted $a \rightarrow b$, where a is the type of the function's argument and b is the type of what it returns. Caliban functions are "curried". So, the type of a function of two arguments, say of types a and b, whose result is of type c, is of type $a \rightarrow (b \rightarrow c)$. Functions may be polymorphic. For example, we might define a function "equals" whose type is described in English by saying that it takes any object and returns a function from objects of the type of that object to booleans. This is represented in Caliban by using type variables (which are just strings of stars). The type of "equals" would be written $* \rightarrow (* \rightarrow \texttt{boolean})$.

Constructed types are built up out of atoms and constructors. Atoms, as their name suggests, are the basic elements of a constructed type. Constructors, as their name suggests, build larger objects of a constructed type from smaller pieces of that type or some other type. Type definitions are introduced with the construct '::='. Consider the definition of NAT:

```
NAT ::= ZERO | SUCC !NAT.
```

This definition says that NAT consists of one atom, ZERO, and one unary constructor, SUCC, which takes an element of NAT as argument and constructs a new element of NAT, namely its argument's successor. The '!' indicates that SUCC is strict in its argument. We note here the existence of the function '#' which maps *num* into NAT. This function makes it possible to represent large natural numbers efficiently: as machine integers of (say) 32 bits rather than as the composition of many applications of the constructor SUCC to ZERO.

In general then, the clauses of a constructed type, separated by '|', describe the means by which objects of that type may be obtained. They may be atoms or constructors, described by giving the name of the constructor followed in order by the names of the types of its arguments, with an optional strictness annotation; or they may name other types.

A new type name can be introduced as a shorthand using the symbol ':: '. Also, an object x is given type T by $x :: T$.

5.3.2 Caliban Expressions

The simplest form of expression is just an identifier or constant. The value of such an expression is the value of the identifier or constant. More interesting expressions are built

up by applying functions to arguments which are expressions. Application is denoted by juxtaposition: $f\,x$. As mentioned above, all functions are curried and application associates to the left. Functions may be built-in, e.g., operations on built-in data types such as *num* or *boolean*, or defined by the user. Both built-in and user-defined functions may be denoted by infixed symbols.

Caliban also provides *guarded expressions*. Guarded expressions allow one to formulate expressions whose value depends on the truth of some condition. The guard is an expression of type *boolean*. The value of

⟨exp1⟩, ⟨guard⟩

is ⟨exp1⟩, if ⟨guard⟩ is true. A guarded expression may be followed by another expression, say ⟨exp2⟩, which is the value of the guarded expression in case ⟨guard⟩ is false. This is written

⟨exp1⟩, ⟨guard⟩; ⟨exp2⟩.

Parenthesis are available for grouping. Indentation may also be used for this purpose, according to the *offsides rule* which states that no character of an expression is to the left of its first character. For example, using the offsides rule the guarded expression immediately above can be written

⟨exp1⟩, ⟨guard⟩
⟨exp2⟩.

5.3.3 Caliban Definitions

Definitions of objects are introduced with = and are made by "pattern matching." For example, we define the functions `map` and `filter` below. If `x` is a list, then `map f x` is the result of applying f to each element of `x`; and `filter P x` is the sublist of `x` consisting of those elements, `a`, such that `P a` is true. Note the use of conditional expressions and the offsides rule in the definition of `filter`.

```
map f [ ] = [ ]
map f (a:x) = (f a):(map f x)

filter P [ ] = [ ]
filter P (a:x) = a:(filter P x) , P x
                 filter P x
```

The definitions of map and filter also use recursion. Recursion in Caliban is unrestricted, and Clio allows one to reason about functions defined with unrestricted recursion. It is also possible to define *infinite* lists since Caliban uses a nonstrict semantics. For example, the following defines an unbounded search operator `least`.

```
least P x = x, P x
            least P (SUCC x)
```

A second-order operator `trace` which generates the infinite list of iterates of an operation `op` is defined by:

```
trace op x = x:(trace op (op x))
```

The semantics of Caliban is provided by a Scott domain, D, which is a complete partial order with a *bottom* element denoted by `bottom`. Symbols (such as `least` and `trace`) denote the least fixed point of their definitions, and the least fixed point always exists.

5.3.4 Clio's Metalanguage

The set of true assertions is the first order theory of the Caliban domain D. The assertion language is first-order predicate calculus over the language of D. A term of this language is a Caliban expression enclosed in back-quotes. For example `` `least P x` `` is a term in the language of D. The atomic formulae are of the form `term1 = term2` and `term1 <= term2`, where `<=` is the partial ordering of the domain D. The formulae are closed under the logical connectives &, \/, ~, => and quantifiers `(x)`, `(Ex)`. For example, here is a true (and provable!) assertion about `least`:

```
assertion1 := (x)(P) (`P (least P x)`='true` &
                          `x <= (least P x)`='true`)
                      \/ `least P x`='bottom`
```

Here the left side, `assertion1`, is an abbreviation, introduced by ':=', of the right side. The back-quotes are necessary for two reasons. The first is that the logical symbols are all overloaded. For example, `trace (~) x` is a Caliban expression in which `(~)` denotes a Caliban operator, which is an element of the domain D of type `bool->bool`. In the language of D, however, ~, is logical negation; the back-quotes distinguish between the two uses. For example, in the assertion `` ~(`f (~x)`='g x`) `` the first ~ is "assertion level not" while the second is "Caliban not". The second reason for the back quotes is that they help us keep in mind the distinction between Caliban expressions like `least P x` and what they denote, `` `least P x` ``, and they remind us of which parts of our assertions are executable Caliban expressions.

The quantifier `(x)` in `assertion1` does not range over the whole domain D. Clio infers a type for every expression. The quantifiers, `(x)` and `(Ex)` are actually bounded quantifiers `(x :: tau)` and `(Ex :: tau)` ranging over objects whose type unifies with tau. In `assertion1` the quantifiers are `(x::NAT) (P::NAT->bool)`.

5.3.5 A Summary of Ada-Ariel Semantics

The meaning of an Ada-Ariel program P is a *dynamic structure* which is a tree of *state structures*. A state structure is a collection of types and functions. The signature of a state structure is fixed both by the language Ada-Ariel and P itself. Certain of the types and functions of a state structure are dynamic: they are allowed to change to produce a new state structure of the same signature. The *transition rules* which govern such changes are also fixed by Ada-Ariel. A dynamic structure is a tree of state structures because transition rules are essentially nondeterministic; the branches at any node of a dynamic structure reflect the choices of next state structure possible at that point in the modelled computation. For brevity, we shall refer to the state structures of a dynamic structure as its states.

The static types and functions of the dynamic structure's states represent the immutable parts of the computation, that is: the program itself; atomic data types such as *integer*, *Boolean*, and *float*; and the operations on these types. The unchanging nature of the representation of a program distinguishes our approach to operational semantics from others. One's intuition is that control moves around the text of a program as it executes and our semantics formalizes this intuition. Atomic data types are fixed by the language and don't change from state to state. The dynamic types and functions of a state represent things which do change: the current locus of control; the history of yet-to-be-completed subroutine calls; and incarnations of program variables and their values.

In order to understand how Ada-Ariel semantics are written down, one must first understand the composition of a state structure, the language used to describe it, and the transition rules that govern its changes. A signature consists essentially of a collection of symbols for functions and basic types along with additional symbols for type forming operations. This language is strongly typed. An interpretation function $[\![-]\!]$ maps symbols from a state structure signature to their denotations. We will provide a brief summary of the functions and types typically composing a state structure in a moment.

Transition rules are written down in the Dynamic Specification Language (DSL, for short). Transition rules are built from *guards* and *updates*. A transition rule consists of a guard and one or more updates. Updates essentially prescribe how dynamic functions or types will be changed in the next state of a dynamic structure. For example, if c denotes a dynamic function and $e_1, \ldots, e_n, e$ are DSL terms, then $c\,e_1 \ldots e_n \leftarrow e$ is an update. Informally, its meaning is: evaluate $e_1, \ldots, e_n, e$ in the current state to obtain $e_1', \ldots, e_n', e'$; then, in the next state, $c\,e_1' \ldots e_n'$ will be e'. A guard is a DSL formula which controls whether an update is applied. If the guard evaluates to *true* in the current state, the corresponding update(s) is(are) applied and otherwise, not. DSL provides a more complicated forms of transition rule. Their description, along with their formal semantics, appears in the Ada-Ariel semantics document [ORA91].

We will now sketch the components of a state structure. We begin with the static components. As mentioned above, a program is represented by its abstract syntax tree. An abstract syntax tree is composed of elements of a static type *Node* which are organized into a tree by the static functions *Begets*, which gives a node's children (if it has any), and *Pre-*

cedes, which orders the children. Nodes are decorated with various static information. Every node has a *label* which is the grammatical category the node represents. Nodes representing identifiers and constants (in the Ada sense) have a *tag* which is the specific identifier or constant denoted by the Ada construct. For example, a node representing an occurrence of the identifier "foo" will have 'ident' as its label, where 'ident' is the name of a grammatical category, and 'foo' as its tag, where 'foo' is an element of the type of identifiers.

A state structure contains static types corresponding to the data types appearing in an Ada-Ariel program. It is from these types that incarnations of program variables take their values. We shall call these types collectively *value types*. The incarnations themselves are represented by values from an analogous collection of dynamic types called collectively *object types*. We will say more about them in a moment. The basic value types include *universal_integer*, *integer*, *universal_real*, *float*, and enumeration types including *character* and *boolean*. Complex value types correspond to array values, which are essentially functions from products of initial segments of the natural numbers to other value types, either basic or complex. This is where the higher order nature of our structures manifests itself.

Static types that encode information internal to the functioning of the Ada-Ariel abstract interpreter are *Outcome*, which represents the result of interpreting an abstract syntax (sub)tree, and *Symbol*, which contains type and constant symbols from the state structure's signature plus other symbols that are used to store type names internally.

The static functions are comprised of those already mentioned in conjunction with the representation of the abstract syntax tree. Also, there are the operations on the basic types. These operations correspond directly to relational, arithmetic, boolean, and other operations available in Ada-Ariel; they need not be inventoried here.

We turn now to the dynamic components of a state structure. Object types are the Ada-Ariel version of what are often called locations. They represent incarnations of program variables and access type values. New elements of an object type are created during declaration elaboration for local variables of the Ada type corresponding to the object type. Elements of an object type are mapped to values by the dynamic function *Store*. Thus, *Store* is the means by which incarnations of program variables are associated with the values stored in them. Object types and the *Store* function constitute an Ada-Ariel interpreter's abstract memory. Note that object types are not linearly ordered and no arithmetic-like operations are defined on them, so our abstract memories are truly abstract.

An Ada-Ariel state structure contains an analog to what is traditionally called the environment in programming language semantics. Its role in our semantics is a combination of its usual one, namely, keeping track of the bindings of program variables to object type values, and an additional role, marking the nodes of an abstract syntax tree with the result of interpreting the subtree of which the node is the root. The target type of the environment function is sequences (used as stacks) of triples consisting of

- a sequence of symbols giving the type of the second element of the triple,

- an element of either a value or an object type,
- an element of the type *Outcome*.

We call such triples *Stack_elements*. *Stack* is an abbreviation for sequences of such elements. *Environment* maps nodes to stacks.

The third component of a stack element will always indicate whether the subtree under the node that is its preimage under *Environment* has been evaluated and, if so, whether evaluation was successful or not. Sequencing through the abstract syntax tree is partially controlled by interrogating this component of the top element in the stacks of the nodes encountered during evaluation. The kind of value in the second component of a stack element depends on grammatical category of its preimage node. If the latter represents a variable declaration, then the second component is a value of the appropriate object type. This is how the object type values that represent incarnations of program variables are associated with the subtrees that represent the variables. If the node is any part of an expression subtree, the second component is a value of the appropriate value type. This how we represent the result of subexpression evaluation in Ada-Ariel semantics.

There are two other dynamic components of state structures that participate in the control of sequencing through an abstract syntax tree. The first of these is the *Active_node*. *Active_node* indicates the current locus of control in the abstract syntax tree being evaluated. In addition, the locus of control in subtrees which have yet to be completely traversed must be recorded. This is necessary for subroutine evaluation and declaration elaboration where the type indication of the declaration references a user defined type by name. In the latter case, the declaration subtree for the named type must be traversed, after which control must return to the original declaration. The *Node_stack* makes this possible. A node stack is a sequence of nodes which is manipulated in stack-like fashion.

5.3.6 A Caliban Encoding of an Ariel Semantics for nanoAVA

Abstract syntax trees are represented as constructed types. The constructors in each of the following Caliban clauses correspond to the "record" names in the Boyer-Moore definition of the abstract syntax of nanoAVA (except as noted).

```
AST ::= Sub AST AST AST AST  |
        NpsList AST AST      |
        Ps AST AST           |
        NbdiList AST AST     |
        BdiVar AST AST AST   |
        BdiConst AST AST AST |
        StmList AST AST      |
        StmNull              |
        StmAsg AST AST       |
```

```
        ExpIde AST           |
        ExpRel AST AST AST   |
        OprEq | OprNe | OprLt | OprLe | OprGt | OprGe |
        Ide
```

A Node in an abstract syntax tree is denoted by the path to it from the tree's root. A path is a sequence of moves; moves are elements of a constructed type with the following definition:

```
MOVE ::= C0  |    || the node itself (useful for technical reasons)
         C1  |    || the node's first child
         C2  |    || the node's second child
         C3  |    || the node's third child
         C4  |    || the node's fourth child
         P        || the node's parent
PATH ::~ [MOVE]
```

The || symbol indicates the beginning of a comment.

To navigate in an abstract syntax tree, we define a function

```
node::AST->PATH->AST
```

We define node by pattern matching: when the path passed it as argument is empty, it returns its abstract syntax tree argument; otherwise, it calls itself recursively on the child of the abstract syntax tree argument given by the first move in the path argement. The following are representative cases of a definition by pattern matching of node.

```
node a [] = a
node (Sub c1 c2 c3 c4) Ci:l = node ci l   || where i is 1,2,3, or 4
```

We also define functions childi, where i is 1,2,3, or 4. These functions are of type AST->AST and map a node to its ith child. The following are representative cases of definitions by pattern matching for the functions childi.

```
child1 (Sub c1 c2 c3 c4) = c1
child2 (Sub c1 c2 c3 c4) = c2
.
.
.
```

There are two static functions defined on an Ariel abstract syntax tree. The first is tag, which maps nodes representing identifiers and constants to the identifier or constant denoted

(an element of DATA). The other static function is intro, which maps nodes representing identifiers to the declaration introducing it. In summary, we have

```
tag::AST->DATA
intro::AST->AST
```

Data values are elements of the following constructed type which is essentially a discriminated union (in the order in which they appear in the type declaration) of integers, the predefined enumerated type Boolean, objects (Ariel locations), and identifiers. Identifiers are present in the data space so that the target type of intro can be DATA, obviating the need to create a special type, consisting of the union of the integers and enumerated types plus identifiers, to serve in this role. The integers are constructed out of the natural numbers (which are a predefined Caliban type); we omit the definition.

```
DATA ::= Int INT |
         Et ENTYPE IDENT |        || enumerated types
         Obj TYPE_SYMBOL NAT |    || object types
         Id IDENT
```

ENTYPE (for enumerated type) is a list of identifiers — the identifiers of the enumerated type, in the order in which they appear in the type's declaration. IDENT is a list of characters. We have

```
IDENT ::~ [CHAR]
ENTYPE ::~ [IDENT]
```

A data value of an enumerated type consists of a representation of the type itself and an identifier. The identifier will appear in the list of identifiers representing the enumerated type. A data value of an object type consists of a type symbol representing the type of the object in question and a natural number which distinguishes the value from other values of the same type. Note that this natural number serves only to distinguish values of the same type; it does not indicate an ordering among them.

Type symbols are tokens that represent types; they occur as components of the stack elements that are attached to abstract tree nodes.

```
TYPE_SYMBOL ::= Integer     |
                IntegerObj  |
                Boolean     |
                BooleanObj
```

Values from the constructed type of outcomes represent whether a subtree has been evaluated and whether or not an error occurred during its evaluation. The definition is as follows.

```
OUTCOME ::= Uneval  |  || unevaluated
            Ok      |  || successful evaluation
            Error      || unsuccessful evaluation
```

As explained in the summary of Ada-Ariel semantics above, the environment function maps abstract syntax tree nodes to stacks of objects which represent either the bindings of program variables or the partial results and outcomes of subcomputations. Because of its simplicity, we will only require one element "stacks" to describe nanoAVA semantics (there is no recursion in nanoAVA). Stack elements have the following definition as a tuple.

```
STACK_ELT ::~ <<DATA, OUTCOME>>
```

We define projection functions on STACK_ELT. They are

```
env_data :: STACK_ELT->DATA
env_outc :: STACK_ELT->OUTCOME
```

We will now describe the components of a state of a nanoAVA computation in Ariel semantics. The components of a state include all the dynamic parts of an Ariel structure. In the case of nanoAVA, a state consists of a node, an environment, a store, and representations of the dynamic universes of integer and boolean objects. The node component represents the currently active node in a program's abstract syntax tree. Its type is PATH. The dynamic function env maps abstract syntax tree nodes to stack elements. The dynamic function store maps objects to values (for technical reasons its type is DATA->DATA). The dynamic universes are represented by natural numbers — the "name" of the next element of the universe to be created. States have the following definition

```
STATE ::~ <<PATH, AST->STACK_ELT, DATA->DATA, NAT, NAT>>
```

We now define the next state function. We first give several useful functions whose use will clarify our presentation. The first is

```
evaluate :: STATE -> MOVE -> STATE
evaluate <<n, e, s, i, b>> m = <<n++[m], e, s, i, b>>
```

evaluate makes active the node obtained by appending its MOVE argument to the path denoting the currently active node; the MOVE argument will denote one of the currently active node's children. The effect is to cause traversal of the subtree rooted at the child in question. The second "utility" function is

```
evaluated :: AST->(AST->STACK_ELT)->BOOL
evaluated e n = IS_Ok (env_outc (e n))
```

`evaluated` takes an abstract syntax tree node and an environment as arguments and checks whether the environment maps the node to a stack element whose outcome component is `Ok`. `IS_Ok` is an example of the predefined predicates Caliban provides for each constructor of a constructed type; it returns true whenever its argument is an element of the type constructed by the constructor. A function `unevaluated` is defined similarly using `IS_Uneval`.

Next, we define an "update" function which will be used to implement, in Caliban, transition rules applied to dynamic functions. The definition is as follows.

```
update f x y z = (x = z) -> y; f z
```

Recall that all Caliban functions are curried. Here, `x` is the argument at which `f` is to change and `y` is the new value of `f x`.

We now proceed to define the next state function, `ns`. The type of `ns` is

```
ns :: AST->STATE->STATE
```

The abstract syntax tree argument to `ns` is the program to be executed; the state argument is the current state of the program's execution. We will define `ns p <<n,e,s,i,b>>` as a nesting of conditional Caliban expressions. The first clause handles subprograms, the largest nanoAVA syntactic unit.

```
IS_Sub (node p n) |->
  (unevaluated e (child3 (node p n))) |-> evaluate <<n,e,s,i,b>> C3
  (evaluated e (child3 (node p n))) &
  (unevaluated e (child4 (node p n))) |-> evaluate <<n,e,s,i,b>> C4
  (evaluated e (child3 (node p n))) &
  (evaluated e (child4 (node p n))) |->
    <<n,
      update (update (update e (child3 (node p n)) <<bottom,Uneval>>)
                     (child4 (node p n))
                     <<bottom,Uneval>>)
             (node p n)
             <<bottom,Ok>>,
      s,i,b>>
```

No clause is necessary for parameter lists because we assume that actual values have been associated with the corresponding formal parameters by the semantic actions for subroutine calls (which are ignored in nanoAVA).

The next clause handles the case where the active node represents a list of statements or declarations.

```
IS_StmList (node p n) | IS_NbdiList (node p n) |->
   unevaluated e (child1 (node p n)) |-> evaluate <<n,e,s,i,b>> C1
   evaluated e (child1 (node p n)) &
   unevaluated e (child2 (node p n)) |-> evaluate <<n,e,s,i,b>> C2;
   evaluated e (child1 (node p n)) &
   evaluated e (child2 (node p n)) |->
     <<reverse (tl (reverse n)),
       update (update (update e (child2 (node p n)) <<bottom,Uneval>>)
                      (child1 (node p n))
                      <<bottom,Uneval>>)
              (node p n)
              <<bottom,Ok>>,
       s,i,b>>
```

The next clause handles variable and constant declarations.

```
IS_BdiVar (node p n) | IS_BdiConst (node p n) |->
   unevaluated e (child3 (node p n)) |-> evaluate <<n,e,s,i,b>> C3;
   evaluated e (child3 (node p n)) |->
     <<reverse (tl (reverse n)),
       update (update e (child3 (node p n)) <<bottom,Uneval>>)
              (node p n)
              <<(Obj IntegerObj iobj),(tag (child2 n))="INTEGER";
                (Obj BooleanObj bobj),
                Ok>>,
       update s
              (Obj IntegerObj i),(tag (child2 n))="INTEGER"
              (Obj BooleanObj b)
              (env_data (e (child3 (node p n)))),
       (Succ i),(tag (child2 n))="INTEGER";i,
       (Succ b),(tag (child2 n))="BOOLEAN";b>>;
```

The next clause handles null statements.

```
IS_StmNull (node p n) |->
  <<reverse (tl (reverse n)),
    update e (node p n) <<bottom,Ok>>,
    s,i,b>>;
```

The next clause handles assignment statements.

```
IS_StmAsg (node p n) |->
```

```
unevaluated e (child2 (node p n)) |-> evaluate <<n,e,s,i,b>> C2;
evaluated e (child2 (node p n)) |->
  <<reverse (tl (reverse n)),
    update (update e (child2 (node p n)) <<bottom,Uneval>>)
           (node p n)
           <<bottom,Ok>>,
    update s
           (env_data e (intro (child1 (node p n))))
           (env_data e (child2 (node p n))),
    i,b>>;
```

The next clause handles expressions consisting of a single identifier.

```
IS_ExpIde (node p n) |->
  <<reverse (tl (reverse n)),
    update e (node p n) <<s (env_data e (intro (node p n))),Ok>>,
    s,i,b>>;
```

The clause for relational expressions must handle each of the six relational operators. These are all straightforward, and are omitted.

5.4 Translating AVA to Ariel

In this section we will describe a translation from the AVA representation of a program to its Ariel counterpart. Since the Boyer-Moore prover, and therefore the AVA system, is so intimately bound up with Lisp, it is most sensible to express an AVA-Ariel translation in Lisp. Among other advantages, doing so eliminates the need to parse the representation, in the language of the Boyer-Moore logic, of the program to be translated; it is, after all, represented as a Lisp form. To quote Boyer and Moore themselves: "We have carefully defined the syntax of terms so that if you type formulas in the indicated syntax they are read properly by Lisp" (*A Computational Logic Handbook*, p.183 [BM88]). We will write our translation in Common Lisp. The translation routines will produce strings which are appropriate input for Clio/Caliban. We will leave these strings in Lisp lists and ignore, for the time being, file input/output.

The translation scheme is recursive descent on the abstract syntax tree. The main translation routine is `trans`, which takes three arguments. The first of these is the piece of the abstract syntax tree to be translated. The second is the path, in the Ariel abstract syntax tree, to the root of the Ariel subtree to be produced by the current incarnation of `trans`. As we shall see below, this path information is necessary to produce a proper definition of the static functions `tag` and `intro`. The third argument is a "symbol table"; it is also necessary for producing a definition of `intro`.

trans produces a list containing four elements. The first element is a string which represents the Caliban translation of the AVA subtree passed as first argument. The second element is a list of dotted pairs used to construct the definition of **tag**. The *car* of each dotted pair in the list is a path in the Ariel abstract syntax tree under construction that leads to an identifier node; the *cdr* of such a pair is the identifier, represented as a string, denoted by the identifier node. The third element is again a list of dotted pairs. In this case, the list is used to create the definition of **intro**. Both the *car* and the *cdr* of each pair in the list are paths. The interpretation is that **intro** maps the node denoted by the path obtained from *car* to the node denoted by the path obtained from *cdr*. The fourth element is once more a list of dotted pairs. In this case, the pairs represent information to be added to the "symbol table". The *cons* of each pair is an identifier represented as a Lisp symbol (as it was encountered in the AVA abstract syntax tree); the *cdr* of each pair is the root of the Ariel subtree where the identifier is declared. We will see below how this "symbol table" information is used.

The main control structure in **trans** is a case macro. An element of an AVA abstract syntax tree is a list of the form

```
(s c1 c2 c3 ... cn)
```

where **s** is a symbol and the **ci** are lists. Our case macro checks **s** and takes appropriate action. Consequently, the definition of **trans** begins as follows

```
(defun trans (x path syms)
  (case (first x)
```

We will now consider each of the cases individually.

The first case handles subroutines, represented by lists of the form

```
(sub ide n-ps-list n-bdi-list stm-list)
```

Now, the third, fourth, and fifth elements of such a list are themselves lists of the kind of object translated by **trans**, but their structure is slightly different — they don't begin with a symbol. So, we have defined separate routines, **n-ps-list**, **n-bdi-list**, and **stm-list**, respectively, to translate them. Each of these routines produces a four element list as its result, as does **trans**. We will give the definitions of these routines below. The current case of **trans** calls the aforementioned routines and stores the results in local variables. It then reassembles the results as its output.

```
('sub (let* (t3 (n-ps-list (third x)
                           (append path (list "C2"))
```

```
                        syms))
          (t4 (n-bdi-list (fourth x)
                          (append path (list "C3"))
                          (fourth t3)))
          (t5 (stm-list (fifth x)
                        (append path (list "C5"))
                        (fourth t4))))
  (list
    (concatenate 'string "Sub " (symbol-name (second x))
                                " (" (first t3) ")"
                                " (" (first t4) ")"
                                " (" (first t5) ")" )
    (append (second t3) (second t4) (second t5))
    (append (third t3) (third t4) (third t5))
    (fourth t5))))
```

The next case handles parameter specifications, represented by lists of the form

```
(nps ide ide)
```

The first identifier is the parameter's name and the second is the name of its type.

```
('nps (list
  "Ps Ide Ide"
  (list (cons (append path (list "C1"))
              (symbol-name (second x)))
        (cons (append path (list "C2"))
              (symbol-name (third x))))
  NIL
  (list (cons (symbol-name (second x)) path))))
```

The next case handles variable and constant declarations, represented by lists of the forms

```
(nbdi-var ide ide exp)
```

and

```
(nbdi-const ide ide exp)
```

The first identifier is the variable or constant's name, the second is the name of its type, and the expression is the variable's initial value or the constant's value.

```
(('nbdi-var 'nbdi-const)
  (let ((texp (trans (fourth x)
                     (append path (list "C3"))
                     syms)))
    (list
      (concatenate 'string
        (case (first x)
          ('nbdi-var "BdiVar")
          ('nbdi-const "BdiConst))
        "Ide" "Ide" (first texp))
      (list (cons (append path (list "C1"))
                  (symbol-name (second x)))
            (cons (append path (list "C2"))
                  (symbol-name (third x))))
      (third texp)
      (list (cons (symbol-name (second x)) path)))))
```

The next case handles null statements.

```
('stm-null (list "StmNull"
                 NIL
                 NIL
                 syms))
```

The next statement handles assignment statements, represented by lists of the form

```
(stm-asg ide exp)
```

The code is as follows

```
('stm-asg
  (let ((texp (trans (third x)
                     (append path (list "C2"))
                     syms)))
    (list
      (concatenate 'string "StmAsg Ide ("
                           (first texp)
                           ")" )
      (append (list (cons (append path (list "C1"))
                          (symbol-name (second x))))
              (second texp))
      (append (list (cons (append path (list "C1"))
```

```
                        (cdr (assoc (second x) syms))))
              (third texp))
      syms)))
```

The next case handles expressions consisting only of an identifier, represented by lists of the form

```
(exp-ide ide)
```

The code is as follows

```
('exp-ide
  (list
    "Ide"
    (list (cons (append path (list "C1"))
                (symbol-name (second x))))
    (list (cons (append path (list "C1"))
                (cdr (assoc (second x) syms))))
    syms))
```

The last case handles relational expressions, represented by lists of the form

```
(exp-rel lhs opr rhs)
```

The code is as follows

```
('exp-rel
  (let ((tlhs (trans (second x)
                     (append path (list "C1"))
                     syms))
        (trhs (trans (fourth x)
                     (append path (list "C3"))
                     syms)))
    (list
      (concatenate 'string "ExpRel (" (first tlhs) ") "
                           (transop (third x))
                           " (" (first trhs) ")")
      (append (second tlhs) (second trhs))
      (append (third tlhs) (third tlhs))
      syms)))
```

This case makes use of the following auxiliary function

```
(defun transop (op)
  (case (car op)
    ('opr-eq "OprEq")
    ('opr-ne "OprNe")
    ('opr-lt "OprLt")
    ('opr-le "OprLe")
    ('opr-gt "OprGt")
    ('opr-ge "OprGe"))
```

Next, we give the translation routines for the parameter, declaration and statement lists. First we have the parameter list routine. It handles lists of the form

```
((nps ...) (nps ...) ...)
```

The code is as follows.

```
(defun n-ps-list (x path syms)
  (if (endp (cdr x))
    (trans (car x) path syms)
    (let* ((tps (trans (car x)
                       (append path (list "C1"))
                       syms))
           (tpslist (n-ps-list (cdr x)
                               (append path (list "C2"))
                               (fourth tps)))
      (list
         (concatenate 'string
                      "NpsList (" (first tps) ") (" (first tpslist) ")")
         (append (second tps) (second tpslist))
         (append (third tps) (third tpslist))
         (fourth tpslist)))))
```

Next we give the routine for declaration lists.

```
(defun n-bdi-list (x path syms)
  (if (endp (cdr x))
    (trans (car x) path syms)
    (let* ((tbd (trans (car x)
                       (append path (list "C1"))
                       syms))
           (tbdlist (n-bdi-list (cdr x)
                                (append path (list "C2"))
```

```
                                (fourth tbd))))
      (list
        (concatenate 'string
                     "NbdiList (" (first tbd) ") (" (first tbdlist) ")")
        (append (second tbd) (second tbdlist))
        (append (third tbd) (third tbdlist))
        (fourth tbdlist)))))
```

Next we give the routine for statement lists.

```
(defun stm-list (x path syms)
  (if (endp (cdr x))
    (trans (car x) path syms)
    (let* ((tstm (trans (car x)
                        (append path (list "C1"))
                        syms))
           (tstmlist (trans (cdr x)
                            (append path (list "C2"))
                            syms)))
      (list
        (concatenate 'string
                     "StmList (" (first tstm) ") (" (first tstmlist) ")")
        (append (second tstm) (second tstmlist))
        (append (third tstm) (third (tstmlist))
        syms))))
```

Now we give the function that produces the definition of tag.

```
(defun tag (arg)
   (if (endp arg)
     NIL
     (cons (concatenate 'string
                        "tag (node prog "
                        (putpath (caar arg))
                        ") = "
                        (cdar arg))
           (tag (cdr arg)))))
```

This function uses the auxiliary function putpath

```
(defun putpath (p)
   (flet ((paux (x)
```

```
            (if (endp x)
              ""
              (concatenate 'string
                           ","
                           (car x)
                           (paux (cdr x)))))
     (if (endp p)
       "[]"
       (concatenate 'string
                    "["
                    (car p)
                    (paux (cdr p))
                    "]" ))))
```

Lastly, we give the function that constructs the definition of `intro`.

```
(defun intro (arg)
  (if (endp arg)
    NIL
    (cons (concatenate 'string
                       "intro (node prog "
                       (putpath (caar x))
                       ") = node prog "
                       (putpath (cdar arg))
          (intro (cdr arg)))))
```

We will now provide an example that gives an idea of how assertions about the AVA semantics of programs can be translated into their Ariel equivalent. The example is to prove that a simple swap program behaves as it should, namely, that it exchanges the values stored in its input parameters. This is expressed in the Boyer-Moore logic as follows.

```
(prove '(equal
          (apply-function
            (e-c (mk-sub 'swap
                         (list (mk-ps '(x y) 'integer))
                         (list (mk-bdi-const '(temp)
                                             'integer
                                             (mk-exp-ide 'x)))
                         (list (mk-stm-asg 'x (mk-exp-ide 'y))
                               (mk-stm-asg 'y
                                           (mk-exp-ide 'temp))))
                 store)
            'x)
```

```
          (apply-function store 'y)))
(prove '(equal
          (apply-function
            (e-c (mk-sub 'swap
                         (list (mk-ps '(x y) 'integer))
                         (list (mk-bdi-const '(temp)
                                             'integer
                                             (mk-exp-ide 'x)))
                         (list (mk-stm-asg 'x (mk-exp-ide 'y))
                               (mk-stm-asg 'y
                                           (mk-exp-ide
                                           'temp))))
                 store)
            'y)
          (apply-function store 'x)))
```

In words, prove that the value of `x` (resp. `y`) according to the store resulting from evaluating `swap` in the initial store `store` is the same as the value of `y` (resp. `x`) according to `store`. The meaning of a procedure is a state transforming function.

Recall that the semantic function `e-c` evaluates the statement list of a function definition in the store resulting from evaluation of the function's declarative item list in the initial store. The manner in which the function is called and, in particular, the manner in which the function's formal parameters are associated with the actual parameters of a call is ignored.

To express the above in Ariel, we must say, in Clio's metalanguage, that the result of running the Ariel interpreter on the swap program in an appropriate initial state is a final state whose store component has the properties mentioned above. Therefore, we need to define a function that "runs" the Ariel interpreter. This is straightforward:

```
run :: AST->STATE->STATE
run p <<n,e,s,i,b>> = <<n,e,s,i,b>>, n=[] &
                                    env_outc (e (node p n)) ~= Uneval
                      run p (ns p <<n,e,s,i,b>>)
```

The idea is, given a piece of abstract syntax tree and an appropriate initial state, iterate the "next state" function until control returns to the root of the tree and the environment has marked the root "ok" or "error".

We will also need projection functions for extracting the environment and store components out of states. They are easily defined by pattern matching.

```
env <<n,e,s,i,b>> = e
store <<n,e,s,i,b>> = s
```

The Ariel abstract syntax tree for *swap* is as follows.

```
swap = (Sub
         Ide
         (NpsList (Ps Ide Ide) (Ps Ide Ide))
         (BdiConst Ide Ide (ExpId Ide))
         (StmList (StmAsg Ide (ExpId Ide))
                  (StmAsg Ide (ExpId Ide))))
```

The *tag* function for swap is the following. Recall that a constructor's first argument is its first child in the abstract syntax tree, its second argument is its second child, etc.

```
tag (node swap [C1]) = ''swap''
tag (node swap [C2,C1,C1]) = ''x''
tag (node swap [C2,C1,C2]) = ''INTEGER''
tag (node swap [C2,C2,C1]) = ''y''
tag (node swap [C2,C2,C2]) = ''INTEGER''
tag (node swap [C3,C1]) = ''temp''
tag (node swap [C3,C2]) = ''INTEGER''
tag (node swap [C3,C3,C1]) = ''x''
tag (node swap [C4,C1,C1]) = ''x''
tag (node swap [C4,C1,C2,C1]) = ''y''
tag (node swap [C4,C2,C1]) = ''y''
tag (node swap [C4,C2,C2,C1]) = ''temp''
```

The definition of intro is as follows.

```
intro (node swap [C3,C3,C1]) = node swap [C2,C1]     || x
intro (node swap [C4,C1,C1]) = node swap [C2,C1]
intro (node swap [C4,C1,C2,C1]) = node swap [C2,C2]  || y
intro (node swap [C4,C2,C1]) = node swap [C2,C2]
intro (node swap [C4,C2,C2,C1]) = node swap [C3]     || temp
```

We now define the initial environment in which swap will be interpreted. Let arid denote the environment that maps every node in swap to <<bottom,Uneval>>. Then define

```
n1 :: NAT
n2 :: NAT
init_env = update (update arid
                          (node swap [C2,C1])
                          (Obj IntegerObj n1))
                  (node swap [C2,C2])
                  (Obj IntegerObj n2)
```

The initial store and representations of the dynamic universes of integer and boolean objects need not be specified so specifically.

```
ii :: NAT
bb :: NAT
init_sto :: DATA->DATA
```

The initial state in which we will have swap interpreted is

```
<<[],init_env,init_sto,ii,bb>>
```

We then define expressions for the environment and store that result from the interpretation.

```
res_env = env (run swap <<[],init_env,init_sto,ii,bb>>)
res_sto = store (run swap <<[],init_env,init_sto,ii,bb>>)
```

The proof obligations are then

```
PROVE
 'res_sto (res_env (node swap [C2,C1]))'
    = 'init_sto (init_env (node swap [C2,C2]))'

PROVE
 'res_sto (res_env (node swap [C2,C2]))'
    = 'init_sto (init_env (node swap [C2,C1]))'
```

[Ada83] ANSI. *The Programming Language Ada Reference Manual*, 1983. ANSI/MIL-STD-1815A.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[BM79] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, New York, 1979.

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[DoD88] United States Department of Defense. *Common Ada Programming Support Environment (APSE) Interface Set (CAIS)*, revision a edition, May 1988.

[FO76] Lloyd D. Fosdick and Leon J. Osterweil. Data flow analysis in software reliability. *Computing Surveys*, 8(3), September 1976.

[Fre84] Stefan M. Freudenberger. *On the Use of Global Optimization Algorithms for the Detection of Semantic Programming Errors.* PhD thesis, New York University, 1984.

[Ger81] Steven M. German. *Verifying the Absense of Common Runtime Errors in Computer Programs.* PhD thesis, Stanford University Dept. of Computer Science, June 1981.

[GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report TR 5, DEC/SRC, July 1985.

[Gri81] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[Gua85] David Guaspari. Toward Ada verification. In *Proceedings of the First IDA Workshop on Formal Specification and Verification of Ada*, March 1985.

[Inc89a] Incremental Systems Corporation. *Functional Capabilities Report: STARS IRIS Standard Development and Application Support*, October 1989.

[Inc89b] Incremental Systems Corporation. *IRIS Tree-Attribute Specification: STARS IRIS Standard Development and Application Support*, November 1989.

[Kle56] S. C. Kleene. Representation of events in nerve nets. In C. Shannon and J. McCarthy, editors, *Automata Studies* Princeton University, 1956.

[L+86] D. C. Luckham et al. Anna: A language for annotating Ada programs. Technical Report CSL-84-261, Stanford University, 1986. Reference Manual.

[LST89] David Luckham, Sriram Sankar, and Shuzo Takahashi. Two dimensional pinpointing: An application of formal specification to debugging packages. Technical Report TR00-2, ORA, April 1989.

[McH84] John McHugh. *Towards the Generation of Efficient Formally Verified Code.* PhD thesis, Institute for Computing Science, Univ. Texas at Austin, 1984.

[OO90] Kurt M. Olender and Leon J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Trans. Software Eng.*, 16(3), March 1990.

[ORA91] ORA Corporation. Final report on semantics. Technical Report 91-01, ORA Corporation, February 1991.

[Pol89] Wolfgang Polak. Predicate transformer semantics for Ada. Technical Report 89-39, ORA Corporation, September 1989.

[SI77] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, January 1977.