79

Compilation of schema objects into flat Ada structures

Adolfo Guzmán

ABSTRACT. A method and its design are presented, which translate schema (flexible SSDL objects) into plain memory structures, like those found in ordinary languages such as C or Ada. The method is elaborated using Ada structures as the target of the compilation process.The produced code is multi-tasked, fast in execution and thrifty in memory use; in addition, creation of new classes could proceed without recompilation: new compiled programs (using the new classes) could exchange objects with old programs (compiled without knowledge of these new classes) without adverse effects.

The user is allowed (but not required) to define for each slot whether it is commonly used (hence, speed of use is key for this slot) or rarely used (hence, economy of storage is key for this slot).

I The problem to solve

To translate schema objects (flexible SSDL objects) into memory structures that store the same information and behavior, but which are significantly faster in access time and economize storage used, and which do not grow in size with repeated uses of them.

Advantages: Speed; independence of proprietary languages.

1•1 Assumptions

(a) All values of a slot are primitive data types (integers, strings,...) or instances. No value is a class.

(b) Slots may have default values.

Our Restriction: Only default values declared before compilation will be recognized. Changing the default value of a class at run time will not affect "old" default values of already created instances. That is, there is no dynamic propagation of default values in the compiled code.

Note: Dynamic changes to default values, as noted above, will not ripple through all members of the class affected. This is equivalent to a redefinition of the default value for a given object and slot as "the value specified (if any) when the object was created." But if the default value changes after birth, we will not propagate it.

Opposite view: The opposite view, dynamic propagation of default values, can be accomplished via a function that visits all members of a given class and updates default values, when the default value changes. For this, each slot needs to have a field called the "reason" or "justification" of the value it has: this field contains the *support set* of that value. If the old default value of slot length is 60, and airplane3.length has value 60, we have to distinguish whether this 60 is a "locally asserted" value (i. e., independent evidence about the length of airplane3 has led us to conclude that its length is 60) or is the default value. The justification in one case is "local" and in the other is "Airplane". Thus, when Airplane.length.default_value changes to 50, only "60's" that have "Airplane" as support will change to 50's, but 60's which have other support evidence will remain 60's.

(c) Default values may contradict each other. The "right" one to use is that of the class "closest" to the instance which is about to have the default value.

(d) A slot is assumed to be single-valued. If the value of a slot is an array of seven elements, it is still a "single" value: ONE array.

(e) Only instances are created during execution. No new classes are added. No new slots are added to existing classes.

file: /home/acosta/doc/dproto/ada-structures/data-structures-for-ssdl-objects.fm
Version 1 (Nov 91); changed Dec 1, 91; *printed Dec 2, 91.* Pages 1-18

That is, the only command that creates new objects is of the form x := new (C), which creates a new instance of class C.

(f) Compiled objects are not permanent: they are not automatically saved to disk (made permanent) by the compilation process herein described. Permanency is not addressed here.

Idea: Sketch how to accomplish this. Introduce primitives (read o file) and (write o file)

(g) Operations on objects that belong to the API are: (new C)†; (destroy C); (putval o s v) which is o.s := v; (getval o s) which is o.s.

Operations on objects that do not belong to the API, but are reserved for internal use (by the interpreter, say) are: (remslot s o). In the above, C is a class, s a slot, o an object, v a value.

(h) The compiler has to produce parallel code, in the sense that could be executed by multiple processors.

Our restriction: Serialize access to data structures at slot level. That is, all operations generated by the compiler should be atomic at the slot level. Only one processor can change the value of a slot of an object; other processors can be working on the same object, but on different slots.

(i) Every variable, data store, and port has to have a type declared, prior to compilation. This is demanded by the strong typing of Ada.

II Representation as flat structures

Each instance of an schema object will be represented as a record structure, comprising several fields. This will allow creation of new instances at execution time: it is just the creation of a new memory structure.

Each slot of a schema object will be represented as a field of a record structure, as follows:

```
type PERSON is
    record
      NAME   : STRING (1 .. 10);
      SS_NUM : INTEGER;
      AGE    : INTEGER;
    end record;
```

This type o representation will be the most commonly used, since it allows direct access to the slot: just by saying PERSON3.NAME, for instance. It is called explicit representation of slots.

2•1 Implicit representation

It turns out that it is convenient to have, in addition to the explicit representation of slots, an implicit one, for new slots. The position of these slots will not be known at compilation time; hence, search is necessary. This is called an "implicit" or search representation. It will be represented by a block of, say, 30 triplets; each triplet is of the form <slot name> <justification> <value>:

```
type PERSON is
    record
      NAME   : STRING (1 .. 10);
```

†. Both notations (new C) and new (C) are used to designate the function new with argument C; this should not cause undue confusion.

```
    SS_NUM  : INTEGER;
    AGE     :INTEGER;
  <slotname 1> <justification 1> <value 1>
  <slotname 2> <justification 2> <value 2>
  ....
  end record;
```

Properties here will be stored lexicographically (in ascending name order). The <justification> field has already been explained in 1•1(b)

Most slots will be stored implicitly, to attain speed in their access; others will be stored explicitly. This documents elaborates further on these and other points.

2•2 Representation of primitive values

Primitive values are those data types already present in the target language, Ada to be chosen as an example. Primitive values (integers, characters, strings, real numbers) are represented directly by declaring its type in the record structure of Section 2•1. For example, a slot age with value an integer.

2•2•1 Slots are single-valued

The assumption 1•1(d) makes life easier for the writer of the compiler. Each slot has a single value. Slots that are normally multi-valued (such as objects_owned or friends) need to hold an array containing several objects [or several friends]. In this manner that slot continues to hold a single value: an array of friends, say.

This assumption makes life cumbersome for the user of SSDL, since she is now responsible for building such array and sticking inside it all the multiple values that are going to appear in an otherwise multivalued slot. I suggest to revise later this assumption and hold the opposite assumption: *most slots are multivalued*, and a few ones (specifically designed) are single valued.

2•3 Representation of values that are objects; first try

Slots having as values other schema objects will contain as value a pointer to that object, as usual. This is accomplished in Ada by an access type. for example, the father slot of a person contains a pointer to another person.

```
type PERSON;     -- incomplete declaration
type PERSON_REF is access PERSON;     -- declaration of type PERSON_REF used below
type PERSON is
    record
      NAME    : STRING (1 .. 10);
      SS_NUM  : INTEGER;
      AGE     : INTEGER;
     FATHER   : PERSON_REF:
....
   end record;
```

Similarly, a slot having as value an airplane will hold a variable (a pointer) of type AIRPLANE_REF, and so on.

The above will produce a multiplicity of pointer types, one for each new class. In C, this multiplicity can be reduced by using the cast operator, forcing the type to one of a few.

In Ada, its static type checking can become very cumbersome and constantly get in the way, as the following example shows.

We want to store in variable x the different pilots of several airplanes; thus, a line of our program may read x := airpl.pilot; x has been declared of type Person_Ref; airpl is a variable of type Airplane_Ref. Now, for airplanes piloted by Persons, that poses no problem; nevertheless, when an airplane is piloted by a member of a subclass of Person, say NasaPilot, TopRatedPilot, CanadianPerson, Lawyer,..., the assignment to x will complain of a type mismatch. The trouble is that the typing structure of Ada does not recognize a subset relation between, say, Canadian and Person. We could try to fix this by using the *subtype* construct, but that is likely to provide only shallow relief, since it is implemented as a constraint or restriction on the range of certain slots.While it is reasonable to deduce what slots (language = English or French; skincolor;...) differentiate Canadian from Person, the set of constrained slots is likely to change when we compare Lawyer versus Person, and change again when we compare TopRatedPilot versus Person, etc. In some cases these "characteristic differences" may be hard, impractical or impossible to find, as in USPresident versus Person.

2•4 Representation of values that are objects; second try

To overcome the problem outlined in Section 2•3, pointers to objects will be of only one kind: OBJECT_REF. A class will be an OBJECT with a parameterized structure. An example explains the approach.

Assume the only classes to be considered are Person [with slots Name, Age, Soc_Sec_Num, Father, Mother], Lawyer [with slots those of Person, plus School, DegreeDate], Pilot [with slots those of Person, plus Hours_of_Flight, TypeOfAirplane] and President [with slots those of Person, plus ContryRuled], where the latter are subclasses of Person. Our data structures will be:

```
type OBJECT;     -- incomplete declaration;
type OBJECT_REF is access OBJECT;     -- declaration of type OBJECT_REF used below
type WHAT_CLASS is (Person, Lawyer, Pilot, President);     -- an enumeration class
type OBJECT is
  record
    NAME : STRING (1 .. 26);
    AGE : INTEGER range 1 .. 120;
    SOC_SEC_NUM : INTEGER;
    FATHER: OBJECT_REF;
    MOTHER: OBJECT_REF;
    case WHAT_CLASS is
      when LAWYER =>
          SCHOOL: STRING (1 .. 26);
          DEGREE_DATE : DATE;
      when PILOT =>
          HOURS_OF_FLIGHT : INTEGER;
          TYPE_OF_AIRPLANE: OBJECT_REF;
      when PRESIDENT =>
          CONTRY_RULED : STRING (1 .. 26);
      when PERSON | others =>
          null;
    end case;
  end record;
```

To define instances of Lawyer and Pilot, we will say

PERS4 : OBJECT (WHAT_CLASS => LAWYER);

PERS6: OBJECT (WHAT_CLASS => PILOT);

In the examples, PERS4.DEGREE_DATE refers to the date in which that Lawyer obtained her degree, while PERS6.HOURS_OF_FLIGHT refers to the hours of flights flown by Person 6. It is illegal to say PERS4.CONTRY_RULED, since Pers4 is not a president; Ada will issue a run time error.

Slots such as FATHER or TYPE_OF_AIRPLANE that point to another object are of type OBJECT_REF. Else, slots contain a primitive type.

A given slot such as TYPE_OF_AIRPLANE contains only information that its value should be a pointer to an object; information that such object should indeed be an airplane is lost and this constraint will not be checked at run time. This is in accordance with producing a run-time code as fast as possible.

2•4•1 Multivalued slots

Slots that could contain several values (such as Children, Friends or Airplanes_Flown) will be an array of the corresponding type of elements:

CHILDREN : ARRAY (1 .. 10) of OBJECT_REF;

COIN_VALUES : ARRAY (1 .. 6) of INTEGER

In the example, Children is a slot that could contain, say, (Fred John Mathews) while Coin_Values could contain (1 5 10 25 50).

For the current version, that is all we need, since it is the user's responsibility to form the array and to stick it in the slot. That is, (put o s v) expects v to be of type ARRAY for multiple-valued slots. For a later version, where the compiler handles the forming of the array, we need another field, the number of elements on the array, as follows:

COIN_VALUES : ARRAY (1 .. 6) of INTEGER;

NUM_COIN_VALUES : INTEGER;

Assume that at a given time, slot COIN_VALUES contains 5, 10, 25; then NUM_COIN_VALUES is 3. Execution of (put o COIN_VALUES 1) will result in the array containing 5, 10, 25, 1, and NUM_COIN_VALUES increased to 4. These more friendly behavior of multiple valued slots could be handled in that later version by the code produced by the compiler, taking this tedious chore off the hands of the user. He would have stated in the SSDL program simply COIN_VALUES := 1 meaning "add the value 1 to whatever COIN_VALUES had already."

2•4•2 Default values

Default values can be defined right in the structure. Example: the default of Age is 30:

AGE : INTEGER range 1 .. 120 := 30; --syntaxis ok?

2•4•3 Pros & Cons of parameterized structures

✓ Fast in access

✓ Efficient in memory utilization. Only the legal slots of a given class occupy space in memory.

✗ Each time a new class is defined, in order to compile it, we should recompile all previously compiled programs that could possible interact (exchange objects) with that containing the new class.† We will call to this recompilation of the whole universe of possibly interacting pro-

grams "wholesale recompilation." It is similar to recompiling the whole SSDL system once a new object has been defined into the current Ontos objectbase.

If two programs only exchange primitive data, then no wholesale recompilation is needed for them to interact. And, of course, programs that use newly created classes but interchange no data with other programs can be safely compiled without wholesale recompilation.

2•5 Representation of values that are objects; third try

From above, wholesale recompilation should be a planned process, done in a day agreed upon by the developers, and subject to control, etc.

In order to diminish the need for wholesale recompilations, another form of representation is introduced, more compact in memory use [not all legal slots of a given class need be present in memory], but slower in access time, since it requires dynamic search [position of a slot in memory is not pre-determined by the compiler, but a run-time search in a small array of slots-value pairs is required]. More importantly, this representation does not require wholesale recompilations, and new classes could be recompiled and new instances fed to old programs (compiled before such classes existed) without adverse results. This is the *implicit representation* already defined in Section 2•1. Example:

```
type PERSON is
    record
        NAME   : STRING (1 .. 10);
        SS_NUM : INTEGER;
        AGE    : INTEGER;
      <slotname 1> <justification 1> <value 1>
      <slotname 2> <justification 2> <value 2>
       ....
      end record;
```

Essentially, a space of, say, 30 slot-value pairs is reserved at the bottom of the space occupied by the slots already known, to be used for "new slots to be defined." The addition of a slot in this space is accomplished by adding its name (a string) and the corresponding value of that slot. Looking for a value of a slot in this space requires searching it. For size 30, linear search is ok. A binary search will be more than enough for practical cases. The definition of PERSON will look like

```
type SLOT_TRIPLET is
   record
     SLOT_NAME : STRING (1 .. 26);
     JUSTIFICATION : OBJECT_REF;
     VALUE : OBJECT_REF;
type PERSON is
    record
        NAME   : STRING (1 .. 10);
        SS_NUM : INTEGER;
        AGE    : INTEGER;
```

---

†. An object o_i created by program p_1 is exchanged with another program p_2 when o_i is passed as parameter from p_1 to p_2.

```
    NUM_OF_NEW_SLOTS : INTEGER;     -- how many triples are occupied below
    NEW_SLOTS : ARRAY (1 .. 30) of SLOT_TRIPLETS;
end record;
```

The above will be enough in case all new slots had values of type OBJECT_REF. Since the value of a slot can be of type integer, real, string, object_ref, and Array, a few more differentiations to type SLOT_TRIPLET are needed, as we shall see later.

With the slot triplets, not only new slots can be added later, but "old" (that is, legal slots when the object was defined) slots that are rarely used could go to the triplet array. The idea is that slots having a known value at definition time, as well as commonly used slots, be given an explicit (predefined) memory space within the structure, while rarely used slots and slots not having initial values should be stored implicitly in the triplet array. If these slots acquire some value at running time, they will occupy a triplet of the triplet array.

This document does not support the justification fields, although the data structures have it. Its addition later is straightforward, but is not contemplated here any further.

2•5•1 Need for wholesale recompilations diminished

With the introduction of the triplet array for storage of implicit slots, the need for wholesale recompilations is diminished: New classes can be defined between wholesale recompilations; newly defined slots simply go "meanwhile" to the triplet array. Eventually, when a wholesale recompilation takes place, these new slots will be relocated to the explicit zone (unless they are declared by the user to be "infrequently used" slots†), and the triplet array of every instance will be always empty after a wholesale recompilation.

2•6 The compilation process

As far as the translation of schema objects into flat structures is concerned, two different compilation processes are identified: wholesale compilation, which occurs when all existing classes are accounted for in the parameterized structure which forms part of the definition of OBJECT, and ordinary (also called incremental) compilation, when some new class is not explicitly part of such parameterized structure.

Initially, as classes are being defined, their definitions form part of a centralized "definition repository" where definitions of all classes reside (this is currently kept in the Ontos object base). At some point in time, it is decided to compile some programs. The compiler begins by collecting all definitions of classes and forming a table of slots versus classes, as in Figure 1•.

	Person	Doctor	Cat	Dog	Airplane
Color	✔	✔	✔	✔	✔
Weight	✔	✔	✔	✔	✔
SocSecNum	✔	✔	✘	✘	✘
Degree	✘	✔	✘	✘	✘
BirthDate	✔	✔	✔	✔	✘

Figure 1• Basic table used for compilation.

†. I am assuming that the user is willing to declare a slot as either "commonly" or "infrequently" used. Should she not bother to do this, the assumption should be that such slot is commonly used, if speed of the run-time executing code is our main concern: that assumption will allocate explicit space for such slots.

From this table a unique parameterized structure similar to that of Section 2•4 emerges.† This structure contains all known classes; compilation proceeds by generating explicit slot representations, as already explained. This is called "wholesale recompilation", because many programs (all programs known so far) are compiled under the influence of that structure, so as to make them as fast as possible.‡ Notice that wholesale recompilation is defined as the compilation of code that has no new classes: no other classes than those defined in the parameterized structure. A wholesale recompilation may span several days, with different spurts of compilation activities, as long as no new classes are added.

One day, the definition of a new class is felt. It is added to the schema present, but it is not added to the now slightly obsolete parameterized structure. Compilation of code using this new class is still possible, but slots of this new class now go to the triplet zone of Section 2•5. The same happens with new slots added to old classes. The compilation under stale parameterized structures is called "incremental" or normal compilation. Normal compilation continues until some other day the system administrator decides to make a new "wholesale recompilation."

The user need not know whether he is doing a wholesale or an incremental recompilation. The system keeps track of this, associating to the parameterized structure a bit with values "current" or "stale".¤ *It is thus possible to produce valid compiled code with stale parameterized structures; this code will not attain its full speed potential, which will be attained after a wholesale recompilation.*

The contents of each cell of Figure 1• are not just "present" or legal versus "absent" or illegal. They also contain: single-valued vs. multi-valued; explicit vs. implicit; name of the inverse slot vs. nil or "no inverse;" default value vs. nil or no default value. The exact storage of Figure 1• is not relevant, as long as we maintain the access functions table(o, s, dis), where dis = legal? | valued? | type? | repr? | inverse? | default?, to access above information. For instance, table (Person, age, valued?) returns SINGLE_VALUED, while table (Airplane, Soc_Sec_Num, legal?) returns FALSE.

The first row of headings of Figure 1• represents all legal classes; its first column of headings represents all existing slots. Each column represents all legal slots for that Class (slots originating from the class itself plus slots originating from super-classes).

2•6•1 Maintenance of the basic table

As we saw, the basic table of Figure 1• is initially formed when a wholesale compilation is called and there are new classes defined since last wholesale compilation. The maintenance of such table, that is, keeping it in sync with the appearance of new classes, can be performed in one of these manners:

(a) by the function newclass (C, name...) that defines a new class. This function could just as well add a new column to Figure 1• as soon as a new class is defined. This is the "data driven" approach, and is my recommendation.

(b) by the function compile (program,...) that produces a new compilation. This function could sweep all the classes just defined until now, compare them against the current basic table, and

---

†. Both the table and the parameterized structure are formed initially at the wholesale recompilation. It is best if both are kept together in a file to be used by the compiler at compile time; in particular, the parameterized structure will be copied and will appear in the structure definition part of the Ada code generated.

‡. That all programs known so far are compiled is the responsibility of the user or the system administrator, nor of the system. That is, the system does not keep track of every program created nor signals "program 69 needs to be compiled but it hasn't".

¤. This bit is used by the compiler to decide to produce triplets occupants; the bit itself does not appear in the generated code.

update such table accordingly. This is the "demand driven" approach, and could be slower than (a).

III Production of distributed code

The possible existence of several tasks requires consideration of the case where two tasks are updating the same object at approximately the same time. Since in Ada context switches (from one task to another) can occur anywhere in the code, it is esssential to guarantee that data structures are not corrupted when an operation to build part of a data structure is interrupted in the middle of its work, in favor of another operation that, unknowingly, is going to operate on the same structure. Problems may arise if this structure is found in an "unstable" or inconsistent situation by the second task.

3•1 Locking at the slot level

The above requires the positioning of a lock at least at the slot level [there are no shared structures across slots]. In other words, each slot operation is going to be atomic; only one process or task at a time can modify a given slot of a given object. Other tasks may work on other slots of the same object, or on other objects. If two tasks wish to work on the same slot of the same object at the same time, one of them has to wait; this waiting is enforced by the slot lock.

The user is unaware of these locks; she just proceeds normally and notices perhaps only a slight delay or nothing at all.

Conceptually, each slot of an object has a slot lock, which is one bit with values 0 = free or not locked; 1 = locked = somebody is modifying this slot. This lock is accessed by two functions, test-and-set, which (indivisibly, in a single machine instruction) tests the lock to see whether is 0 and sets it (to 1), and reset, which resets the lock (to 0).

These functions are only used inside two other functions, get-the-lock and release-the-lock. The first tries as many times as necessary until it "gets the lock" (the lock was free and gets-the-lock sets it to locked). The second releases the lock (resets the lock to free).

These two functions, get-the-lock and release-the-lock, are not used everywhere. They are only used at the beginning and at the end of *critical sections*: sections of code whose execution should be free of interruption by other tasks. A critical section should be as short as possible; it is better if there only appears one get-the-lock near the entry point and only one release-the-lock near the exit point.

The code for get-the-lock could look like

```
get-the-lock (lock);
{ while (test-and-set (lock) .not equal. 0) do
        { ; --nothing; or k := k+1 to kill time tracking the number of attempts }
-- once you succeed in getting the lock, that is all! Or you could return 1, the value of the lock.
}
```

The code for release-the-lock is simply to set it to 0, using reset.

I am not aware of an indivisible test-and-set operation in Ada. If it exists, it should be used. If not, it could be approximated by

```
test-and-set (lock);
{ if (lock == 0) { lock := 1; return 0 }
 else {return 1}
```

}

Notice that get-the-lock may spend considerably time *spinning* (testing and testing and testing ...) until it gets the lock.

IV Permanent structures needed for compilation

As we saw in II, three permanent structures are needed for compilation:

(a) the basic table of Figure 1•;

(b) the parameterized structure of Sec. 2•4, initially formed at wholesale compilation time;

(c) the information (Sec. 2•6) about whether the parameterized structure is current or stale.

It is best to keep these structures in a file associated with the SSDL object base from which its information was derived. The format of this file could be

<flag of stale or current>

<parameterized structure>.

<basic table>

4•1 Format of the stale flag

In that file, which should be ascii for clarity, the stale flag could be a string in the first line (record) of such file, with one of two possibilities:

CURRENT -- meaning that the parameterized structure is still current;

STALE -- the parameterized structure no longer reflects all known classes.

This has the advantage that if other file is read (by mistake, say), then an error could be detected and flagged "illegal file containing permanent structures" because the first line is neither CURRENT nor STALE.

4•2 Format of the parameterized structure

The format has been described in Sec. 2•4. A few slots common to all instances are: name (of the instance); member_of (Class of which this instance is a member); creation_time. Other common slots should be identified and factored out of the case statement.

4•3 Format of the basic table

BASIC STRUCTURE BEGINS -- To indicate that a basic table is about to be defined

.... followed by as many records as needed to define the basic structure

END BASIC STRUCTURE -- followed by this line to denote end of the basic structure.

The records comprising the "body" of the basic structure are of the form:

n -- an integer denothing how many lines containing names of classes follow

PERSON TEXAS_PERSON PILOT LAWYER -- n lines containing the names, separated

SCREWDRIVER NAIL HAMMER -- by spaces, of all existing classes

The order of the above strings is immaterial. No names should be missing, duplicated, or broken among two lines of the file.

Then, classes definition follow. A typical class definition will look like

CLASS PILOT -- the definition of this class, PILOT, is about to begin

SLOT age SINGLE_VALUED EXPLICIT nil 30 -- word SLOT, followed by the name of the slot, followed by SINGLE_VALUED or MULTIPLE_VALUED, followed by the name of the inverse of AGE, which is nil (no inverse) in this case, followed by the default value 30.

SLOT Soc_Sec_Num SINGLE_VALUED EXPLICIT nil nil -- another name and 4 values

...

CLASS LAWYER -- PILOT has finished, now definition of class LAWYER begins

....

V Forming the structures

First, the basic table is formed; then, the parameterized structure, and finally, the stale flag. Nevertheless, the order of these structures in the permanent file is the opposite, as shown at the beginning of Sec. IV.

5•1 Forming the basic table

The basic table (Figure 1•) is formed by sweeping all known class definitions existing in the object base. Each new class definition adds a column to the basic table. Each new slot adds a row to it. When finding the legal slots for a given class, attention should be paid to the slots inherited from older (bigger) superclasses, in addition to the slots defined in the class itself.

A given in-memory representation should be selected for the basic table, to be used by the program that forms it, as well as by the program that prints the permanent version of it escribed in Sec. IV. In this section, we describe the formation of the table without reference to a concrete memory structure for it. We only use an abstract representation (that of Figure 1•): a 2-dimensional array where the columns are classes and the rows are slots. Each intersection (slot, class) provides storage to the following information (Cf. Sec. 2•6):

legal? = TRUE when that slot is a valid slot for that class;
= FALSE when that slot is not a valid slot for that class; if the value is FALSE then values associated to other information in this intersection will be ignored.

valued? = SINGLE_VALUED when this slot can only have one value for this class. Example: slot age is single-valued.
= MULTIPLE_VALUED when this slot can have several values. Example: slot friends is multiple-valued.

type? = the class of which the value should be an instance of. Example: slot AGE should be of type INTEGER; slot friends should be of type PERSON_REF. A local definition of a slot may give it a certain type, and an inherited definition of same slot a different type. In that case, the local definition overrides. Same is true when both definitions are inherited: the closer (more local) one overrides.

repr? = EXPLICIT when this slot will have an explicit memory representation.
= IMPLICIT when this slot will have an implicit memory representation.

inverse? = the name of the inverse slot, if this slot has an inverse. Examples: the inverse of slot friends is slots friends_of; the inverse of slot children is children_of. If o.s = v, and s has an inverse s', then v.s' = o, always.
= NULL if this slot has no inverse. If the value of slot s is a primitive data (as opposed to a pointer to an object), then the slot should have no inverse.†

default? = the default value for this slot, if one exists. It has to be of the type specified above in type? When collecting default values, proceed from locally defined ones (those defined in *this* class), and move up superclasses only as far as needed to find the closest default value, if any.

= NULL when no default value is specified.

The function that forms the basic table is:

```
form-basic-table (ta);
{ for each cl in existing-class            -- cl is one of the existing schema classes
    { add-column (cl, ta);                 -- add column LAWYER to table
      for each sl in cl                    -- sl is one of the slots valid for the class (not coming from the super-
                                              classes); these could be called immediate slots, being the slots defined for the class.
        { store-props-in-ta (sl, cl, sl, cl); } --get a bunch of properties from the schema definition of
                                              sl, cl and store them in sl, cl.
        for each acl in ascending-super-classes(cl); -- acl is a class in an ordered list of ascending
                                              super-classes of cl; ordered from closest to farthest superclass
        { if (acl is a member of ta) { copy_row (acl, cl); --copy all the undefined intersections of
                                              row cl from row acl; leave unchanged (don't overwrite) those
                                              intersections of cl that are already defined.
                                       break; -- get out of the for each acl; you already have all your legal
                                              slots
                                     }
          else { store-props-in-ta (sl, cl, sl, acl); -- store in ta (sl, cl) the properties of (sl, acl) which
                                              are still undefined in ta(sl, cl)
               }                       -- continue with next superclass acl
        }
    }
}
```

The function store-props-in-ta (sl, cl, sl_2, cl_2) stores in ta(sl, cl) the properties of (sl_2, cl_2) which are still undefined in ta(sl, cl). The properties of sl_2, cl_2 are obtained from the schema definitions of the classes.

```
store-props-in-ta (sl, cl, sl2, cl2);  -- sl2, cl2 is the donor; sl, cl the recipient
    { if (sl does not exist in the first column of ta) add-row (sl, ta);
      ta (sl, cl, legal?) := TRUE;
      if (ta (sl, cl, valued?) != NULL)  ta(sl, cl, valued?):= get (cl2, sl2, valued?);  -- store SINGLE
                                              or multiple valued
      if (ta (sl, cl, type?)!= NULL)  ta(sl, cl, type?) := get_type (cl2, sl2)  -- store type of slot sl
      if (ta (sl, cl, repr?) == NULL)) ta(sl, cl, repr?) := get_repr (cl2, sl2); -- store slot representation
                                              (explicit/implicit) unless this information already there
      if (ta (sl, cl, inverse?) != NULL)  ta(sl, cl, inverse?):= get_inverse (cl2, sl2); -- store if this slot
                                              has an inverse
      if (ta (sl, cl, default?)!= NULL)  ta(sl, cl, default?):= get-default-value (cl2, sl2);
```

†. The code produced by this compiler does not automatically maintain the values for inverse slots, when these exist. The changes are straightforward but are not contained in this document.

```
                                                --store default value, if any
}
```

5•2 Formation of the parameterized structure

This structure should be formed once all the classes have been found, whenever a new wholesale recompilation is in progress. The user usually says "compile", but occassionally she (or, most likely, the system administrator) can say "wholesale compile." In this case a wholesale compilation is in order. If the stale flag is still CURRENT, nothing special should be done, and we could proceed to ordinarily compile, since the parameterized structure (and the basic table) is still current.

Nevertheless, if the stale flag is STALE, a new wholesale recompilation is about to begin, since there are new classes to be taken into account. First, a new basic table should be computed. From it, the parameterized structure should be formed. Finally, the stale flag should be set to CURRENT.

We shall proceed to describe, thus, the formation of the parameterized structure, from the basic table.† The function form-par-struc (ta) forms such param. structure, starting from table ta (Figure 1•).

```
form-par-struc (ta);
{ form-aux-struc (ta);      -- forms auxiliary structures WHAT_CLASSES and SLOT_TRIPLETS
  form-invariant-part-of-par (ta);    -- forms the "fixed" part of the par. structure
  printf ("case WHAT_CLASS is \n");
   for each column in ta                          -- form structures for each class
     { form-each-class-struc (column, ta); }
   printf (" end case \n");
   form-epilogue-of-par (ta);                  -- form bottom part of structure
}
```

5•2•1 Forming the auxiliary structures

The auxiliary structures WHAT_CLASSES and SLOT_TRIPLETS need to be formed prior to the formation of the main parameterized structure, since they will be used in this.

To form the structure what-classes, we need to collect all legal classes. That is the topmost row of ta. To form the structure SLOT_TRIPLETS (Sec. 2•5) a constant text is produced.

```
form-aux-struc (ta);
{printf ("type WHAT_CLASS is (%s) \n", all-classes-of (ta));    -- Refer to Sec. 2•4
 printf ("type SLOT_TRIPLET is \n  record  \n SLOT_NAME :  STRING (1 .. 26); \n
              JUSTIFICATION:  OBJECT_REF; \n
            VALUE : OBJECT_REF; \n");   --prints the SLOT_TRIPLET structure of Sec. 2•5
}
```

†. The order of computation of the constituents of the permanent file is basic table -> param. structure -> stale flag; the order of appearance in the file is the reverse. The main reason to leave the basic table at the bottom is that it will grow as new classes are added, while the param. structure remains unchanged.

5•2•2 Forming the fixed part of the par. structure

To form the fixed part of the par. structure, we need to collect all slots that are common to every class. These correspond to rows in ta (Figure 1•) having all ✔. For each, we need to form its explicit representation.

```
form-invariant-part-of-par (ta);
printf ("type OBJECT is \n  record");
{ for each sl in common-slots-of (ta)     --common-slots-of (ta) finds common slots and also marks each
                slot as explicit or implicit, as explained in Sec. 2•6. This will be useful in form-each-class-struc.
    { form-representation (sl, PERSON, ta, explicit); }          --this gets repeated for each sl
        -- form-representation needs as argum the class; here, any class (for ex. PERSON) is good enough.
}
```

To form the (explicit/implicit) representation for a slot, the following is enough:

```
form-representation (sl, cla, ta, ex-im);      -- ex-im is EXPLICIT, IMPLICIT or NULL
   -- sl is the slot, cla the class; ta the basic table
{ if (ex-im == NULL) ex-im := ta (sl, cla, repr?)   -- ta (sl, cla, repr?) gives the representation of such slot
    for such class (ref. Section 2•6).
 if (ex-im == NULL) ex-im := DEFAULT_REPR;        -- default-repr is usually EXPLICIT
 if (ex-im == EXPLICIT) {
    printf (" %s : %s ", ta(sl,cla,name?), ta(sl,cla,type?));  -- prints AGE : INTEGER
    if (ta(sl,cla,default?) != NULL) printf (" := %s", ta(sl,cla,default?));  --prints := 17†
    printf ("; \n");
    }
 if (ex-im == IMPLICIT) {  --implicitly stored slots do not occupy an initial space, unless they have initial value
    if (ta(sl,cla,default?)!=NULL) printf ("put0 ( %s, %s, %s); \n", sl, cla, ta(sl,cla,default?));
    -- put0 (age, PERSON, 30) is printed; this executable code is still part of the 'data declaration' part; it may be
    intermixed with other data declarations. It is best if all executable statements are moved to the end of the data
    declaration part, next to the beginning of the executable code. This could be achieved easily, for instance, by
    printing them not on the standard output, but on out1; later, file out1 is appended to the end of the "pure" data
    declarations.
    } --form-representation needs to be changed slightly to take into account single vs multiple values; as it is now
    it only considers single-valued slots -- that's ok for the current version
}
```

5•2•3 Forming each class structure

To form the structure for each class, its slots are scanned and printed (represented).

```
form-each-class-struc (col, ta);     -- col is the column, which is the class
{ printf ("when %s => \n", cl);     -- prints when LAWYER =>
  for each row sl in col                    --row sl is the slot
     { form-representation (sl, col, ta, NULL); } -- This forms the representation for such slot
}
```

---

†. Thus, the total definition is AGE: INTEGER := 17; I am not sure if this is the right syntax to specify initial (defualt) values.

5•2•4 Forming the epilogue of the par. structure

To form the bottom part of the structure, the case needs to be closed, and the array of triplets (Sec. 2•5) needs to be formed.

```
form-epilogue-of-par (ta);
{ printf ("when others => \n    null;  \n    end case;");     -- this finishes the case statement
  printf ("NUM_OF_NEW_SLOTS : INTEGER := 0; \n");
  printf ("NEW_SLOTS : ARRAY (1 .. 30) of SLOT_TRIPLETS; \n");
  printf ("end record; \n");
}
```

The structure SLOT_TRIPLETS was already formed (Sec. 5•2•1).

5•3 Formation of the stale flag

As soon as the parameterized structure has been formed (Sec. 5•2), the stale flag should be set to CURRENT.

As soon as a new class is defined, the stale flag should be set to STALE. As suggested in Section 2•6•1, this maintennance work could be done by the function newclass.

VI Generation of code

With the help of the permanent structures of Sec. IV, SSDL code accessing flexible schema objects can be converted into Ada code accessing memory structures, as follows.

6•1 Class definition

Currently, there is no SSDL statement to define new classes; these are defined with the help of the Schema editor, a menu-driven tool. For the purpose of discussion, let us postulate a SSDL statement new_class (Cnew, C....) for creation of class Cnew as a subclass of (old) class C. The compiler, upon encountering new_class (...), should proceed to call the existing functions (for instance, the function newclass of Sec. 2•6•1)that define such schema class, but otherwise ignore the class definition. These functions have the added duty (See Sec. 2•6•1) of keeping the basic table ta current.

The (new, postulated) SSDL statement new_class (Cnew, C, ...) produces no compiled code; its effects will be observed when producing code for statements that instantiate class Cnew.

6•2 Class instantiation

A typical SSDLstatement for creation of instances of a new class C is

```
x := new (PERSON, name = "Guzmán", age = 30);
```

The compiler should produce the following structure instantiation:

```
x := new OBJECT_REF (WHAT_CLASS => PERSON, NAME = "Guzmán", AGE => 30, ...);
                    --that is all. OBJECT_REF has been defined in Sec. 2•4.
```

6•3 Class destruction

There is no class destruction at run time.

6•4 Instance destruction

The SSDL statetement

destroy (o);

where o is an object which is not a class, will be compiled into

IF (KILLABLE (O)) KILL (O); --kill is the Ada destructor for structure o; killable is TRUE if o is an object which is not a class. Killing an object will produce havoc in other processes trying to access or modify it; it should be used with caution.

6•5 Storing a value in a slot

The SSDL statement

o.s := v

which assigns value v to slot s of object o, will be compiled into

a compiler error; if ta (s, class(o), legal?) == FALSE. --illegal slot for this class

a compiler error; if ta (s, class (o), type?) != type(v) assumming that type(v) can be detected at compile time. --assignment to slot of wrong type:

IF (TYPE (V) != "constant") { RUNTIME-ERROR } --assignment to slot of wrong type

ELSE { O.S :=V | put0(o,s,v) }; if the type of v can not be detected at compile time. The constant is the type given by ta(s, class(o),type?). After the ELSE either one of these two statements o.s :=v; put0(o,s,v) is generated, according to whether the value of ta(s, class(o), repr?) is == EXPLICIT or not.

If the type of v can be detected at compile time, and is equal to ta(s, class(o),type?), then the code generated is:

o.s := v; if ta (s, class(o), repr?) == EXPLICIT.

put0 (o, s, v); if ta (s, class(o), repr?) == IMPLICIT.

6•5•1 The function put0

put0 is a function that must make room in the appropriate place of the array of triplets, for a new triplet of the form <name of slot> <justification> <value v>. Remember that the array of triplets maintains its triplets ordered by slot name. Thus, put0 must do some pushing aside of triplets, in general, to accomodate the pending triplet, in case s was not yet in the array of triplets.

If a linear search is used in the array of triplets, then put0 must place its triplet in the first empty triplet of the array, which is indicated by 1 + NUM_OF_NEW_SLOTS (Sec. 2•5). In this case, the code for put0 is

```
put0(o, s, v);
{ get-the-lock (o, s);
  indx := search(o.new_slots, s);     -- returns the index of the triplet containing s, or -1 if not found
  if (indx == -1) indx := 1 + o.num_of_new_slots;
  o.new_slots[indx].slot_name := s;   -- assign the slot name s;
  o.new_slots[indx].justification = 0; -- local justification;
  o.new_slots[indx].value = v;   -- assign value v;
  o.num_new_slots := indx;
  release-the-lock (o, s);
}
```

6•5•2 The function put4

In the case the type of v can not be detected at compile time, compilation of the SSDL statement o.s := v generates

(a) an expression of the form

```
IF (TYPE (V) != PERSON) { run-time-error ("wrong type: %s is not a %s", v, PERSON); }
ELSE { O.S := V; }
when ta (s, class(o), repr?) == EXPLICIT.
```

(b) An expresion of the form

```
IF (TYPE (V) != PERSON) { run-time-error ("wrong type: %s is not a %s", v, PERSON); }
ELSE { PUT0 (O, S, V); }
when ta (s, class(o), repr?) == IMPLICIT.
```

These expressions are often called put4, and instead of generating (a) or (b), just the code

```
put4 (o, s, v)
```

is generated. We prefer to generate either (a) or (b) inline; that is, put4 is generated open (in-line).

6.5.2.1 The function search

This function looks for a slot in the array of triplets, returns -1 if not found or the index of the array where it is.

```
int search(arr, s);                    -- arr is an array of triplets; s a slot
{ for i in (arr'first .. arr'last)
    { if (arr[i].slot_name == s) return (i); }
 return (-1);   --not found
}
```

6•5•3 Handling the lock at the slot level

In the code generated above, two places need to be protected inside a critical section. One is o.s := v. But this assignment is usually carried out in one machine instruction, so the protection is not needed. The other is put0 (o, s, v). One of the first instructions of put0 must be to get the lock (Sec. 3•1), one of its last statements, to release the lock. Since get0 and put0 use the array of triplets, the lock for this array should be unique --not a lock for each slot, as was the case for the explicitly stored slots.

Where is the lock corresponding to an object and a slot? Ideally, it should be in an array of bits, one for each slot of the object. Such array must form part of the parameterized structure created in Sec. 5•2. This document does not explicitly show how to form this array of bit locks.

6•6 Getting the value of a slot

The SSDL expression

```
o.s
```

which produces the value of slot s of object o, will be compiled into

```
NULL;      if ta(s, class(o), legal?) == FALSE. -- this is more convenient than giving an error
o.s;       if ta(s, class(o), repr?) == EXPLICIT.
get0 (o, s) if ta(s, class(o), repr?) == IMPLICIT.
```

6•6•1 The function get0

This function returns the value that a slot has, or NULL.

```
geto(o, s)
{ i:= search(o.new_slots, s);
 if (i == -1) { return NULL; } else { return new_slots[i].value; }
}
```

The function search has already been explained in Sec. 6.5.1.1.

6•6•2 Handling the lock at the slot level

Like in the previous ssdl statement o.s := v, the only function needed to be protected by the lock is get0(o, s), which must search the array of triplets for the value corresponding to slot s.

6•7 References

[1] Henry Ledgard. ADA - An Introduction. Second Edition. January 1983. Springer-Verlag.