



DTIC

ELECTE

MAY 14 1991

User-Defined Data Types in the State Delta Verification System (SDVS)

Prepared by

J. V. COOK and J. E. DONER
Computer Science and Technology Subdivision

30 September 1990

Prepared for

SPACE SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
Los Angeles Air Force Base
P. O. Box 92960
Los Angeles, CA 90009-2960

Assessment for
NTIS 000001
DTIC Lab
Unannounced
Justification

By
Distribution

Availability Codes

Dist Avail and/or
Special

A-1

Engineering Group

THE AEROSPACE CORPORATION

El Segundo, California

DTIC FILE COPY

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

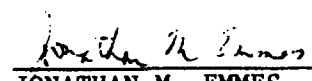
91 5 13 068

This report was submitted by The Aerospace Corporation, El Segundo, CA 90245, under Contract No. F04701-88-C-0089 with the Space Systems Division, P. O. Box 92960, Los Angeles, CA 90009-2960. It was reviewed and approved for The Aerospace Corporation by C. A. Sunshine, Principal Director, Computer Science and Technology Subdivision. Mike Pentony, Lt, USAF, was the project officer for the Mission-Oriented Investigation and Experimentation (MOIE) Program.

This report has been reviewed by the Public Affairs Office (PAS) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nationals.

This technical report has been reviewed and is approved for publication. Publication of this report does not constitute Air Force approval of the report's findings or conclusions. It is published only for the exchange and stimulation of ideas.


MIKE PENTONY, Lt, USAF
MOIE Project Officer
SSD/SDEFS


JONATHAN M. EMMES, Maj, USAF
MOIE Program Manager
PL/WCO OL-AH

CONTENTS

1 Abstract Data Types	3
2 SDVS Implementation	11
3 Conclusions	13
References	15

1 Abstract Data Types

Good programming style calls for dividing the programming effort into manageable submodules, each of which performs separate and clearly defined tasks. Good mathematical style calls for building a structure of intermediate results and lemmas, so as to render the proof of a major theorem clear and ideally, short. In much the same way, the verification of programs calls for the division of a problem into manageable subtasks. Here, the divisions are liable to be suggested by, and operate parallel to, similar divisions made in the programs.

An important aspect of the structuring and subdivision of computer programs is the definition of abstract data types. These provide a system that organizes and accesses information according to prescribed procedures, with a concomitant enhancement of the clarity and comprehensibility of the program. For example, a programmer can define records of various kinds and access them according to named subfields. The superiority of this approach to one in which all data are lumped together in an amorphous mass is obvious, and is taught in all computer science courses.

The paradigm for the use of abstract data types is this: the structure of the data type is defined, as are some functions for accessing it. The specifications of the data type and its supporting functions avoid reference to implementation details, such as whether pointers are used, where within records certain fields are found, and so on. A separate module is created for the purpose of implementing the abstract data type and its supporting functions. Having created the implementation, the programmer elsewhere avoids reference to its details, manipulating the data type and using the supporting functions only according to the abstract, implementation-free, specifications.

Modern programming languages are designed to encourage the programmer to follow this paradigm. Pascal, for example, provides facilities for defining data types and limits access to them to predefined methods. C also has flexible definition facilities, but it does not constrain access methods. More recent languages such as Ada and Modula-2 offer the ability to define "packages" containing data type implementations, while "exporting" only the abstract part of the definitions.

We give two illustrations of abstract data type definitions. The first is

of a stack of integers:

```
typedef stack
  push:  [int × stack → stack]
  pop:   [stack → stack]
  top:   [stack → int]
  emptyStack:  stack

axioms (let s:stack, x:int)
  top(push(x,s)) = x
  pop(push(x,s)) = s
  push(x,s) ≠ emptyStack
  s ≠ emptyStack → s = push(top(s),pop(s))
```

In addition, there need to be some settings of default or error values—it is not always clear what the correct values should be, so let us just suggest some possibilities:

```
top(emptyStack) = 0 ?    = error ?
pop(emptyStack) = emptyStack ?
```

Depending on the implementation, a `newStack` function for allocating new instances of the `stack` data type, and an `isStack(...)` predicate to recognize variables of type `stack`, could be necessary.

There are two obvious models for the stack structure, in which the axioms are satisfied:

1. Stacks are sequences; `push`, `pop`, `top` affect first elements and `⟨⟩ = emptyStack`.
2. Stacks are terms in a formal language that includes function symbols

emptyStack and push (and others to represent integers):

```

emptyStack
push(n, emptyStack)
  ⋮
push(n0, push(n1, push(..., push(nk, emptyStack)...)))
  ⋮

```

The existence of the model shows that the axioms are consistent.¹

Not mentioned yet in this example is the fact that stacks are *finite* data structures. The finiteness property is important because it justifies inductive proofs; in fact, the assertion of finiteness is from some points of view just a way of saying “induction works”. We can suggest two good ways to reflect the property of finiteness within a formal proof system, which in the case of the stack data type example are as follows:

- One can add a new function that measures the size of the finite stack and appropriate axioms for it:

```

size: [stack → int]
size(s) = 0 ↔ s = emptyStack
size(push(x, s)) = size(s) + 1

```

- Alternatively (or in addition), one can introduce a new induction rule that expresses structural induction on stacks:

$$\frac{\varphi(\text{emptyStack}) \quad \varphi(s) \rightarrow \varphi(\text{push}(x, s))}{\varphi(t)}$$

Here, φ represents any well-formed sentence, t is any term, and the variable s does not occur free in any active hypothesis in the proof

¹Formal verification systems have to confront the problem that if the user is allowed to specify the axioms, there is the possibility that those axioms might not be consistent. An alternative is to allow only an automatically generated set of axioms whose consistency can be guaranteed.

leading to $\varphi(s) \rightarrow \varphi(\text{push}(x, s))$. This rule is all that is required in most proofs about stacks that do not reference the implementation.²

Our second example is the binary tree data structure:

```
typedef binaryTree
  emptyTree: binaryTree
  buildTree: [binaryTree × int × binaryTree → binaryTree]
  left:      [binaryTree → binaryTree]
  data:      [binaryTree → int]
  right:     [binaryTree → binaryTree]

axioms (let r,s,t: binaryTree, i: int)
  emptyTree ≠ buildTree(r,i,s)
  left(buildTree(r,i,s)) = r
  data(buildTree(r,i,s)) = i
  right(buildTree(r,i,s)) = s
  t ≠ emptyTree → t = buildTree(left(t),data(t),right(t))
  left(emptyTree) = emptyTree
  data(emptyTree) = 0
  right(emptyTree) = emptyTree
```

Here again we recognize two not-very-different ways to construct models for the axioms:

1. The least class of sequences containing $\langle \rangle$ and containing any sequence $\langle r, i, s \rangle$, where i is an integer and r, s are sequences in this class;

²The form given for the induction rule is one typically used by proof-theorists in discussing quantifier-free systems. A more typical form for other systems would be

$$\frac{\varphi(\text{emptyStack}) \quad \forall s[\varphi(s) \rightarrow \varphi(\text{push}(x, s))]}{\forall s \varphi(s)}$$

We use the first form here because it is simpler. An actual implementation in SDVS would look rather different from either form, but it would be more closely related to the first.

2. The class of terms built from `emptyTree` and terms for integers using the `buildTree` function symbol.

Finiteness can be expressed in ways similar to those for the `stack` data type:

- First, with a `treeSize` function:

```
treeSize: [binaryTree → int]
treeSize(emptyTree) = 0
treeSize(buildTree(r,i,s)) =
    treeSize(r) + 1 + treeSize(s)
```

- Second, with a principle of tree induction:

$$\frac{\varphi(\text{emptyTree}) \quad \varphi(s) \ \& \ \varphi(r) \rightarrow \varphi(\text{buildTree}(r,i,s))}{\varphi(t)}$$

where φ is any sentence, i is a variable of type `int` and does not occur free in φ , and r, s, t are variables of type `binaryTree`.

The tree example brings us clearly to grips with the abstractness of the data type definitions. A typical implementation of binary trees uses a record with a field for the data and two fields holding pointers to other binary trees (the left and right subtrees). However, the abstract specification above makes no mention of pointers, so that we have a binary tree data structure in pure form. One should conceive of our specification as that which is exported from the module implementing the binary tree data structure.

The verification problem for programs containing user-defined data types that implement abstract data type definitions in the way we have described divides into two phases. First, it is necessary to prove that the implementation module for the data type correctly implements it; this means that one must prove that the axioms hold for the structure as implemented. In the second phase, one proves the correctness of the program using the defined data type, when this program uses only those aspects of the data type that are exported from the definition module. This proof should employ only the axioms given, along with the type information about the supporting functions for the data type, and should make no reference to the implementation.

The first phase, proving implementation correctness, is likely to be more difficult, especially in view of the need to deal with pointers. The second phase, proving the correctness of the higher-level program, seems more tractable and is the subject of this report. We suggest that a data type definition facility be added to SDVS, and we describe the other features that must be added to support this facility.

In their program for Computational Logic [1], Boyer and Moore developed a formal method for introducing new user-defined data types, called the *shell principle*. This specifies what functions are introduced for the definition of a user-defined data type, and introduces an automatically generated set of associated axioms. The following is the pattern of the shell principle:

There is a *constructor* function **const**, of **n** arguments;
 an optional *base constant* **base**;
 a *recognizer* function **r**;
accessor functions **ac₁**, **ac₂**, ..., **ac_n**;
type restrictions **tr₁**, **tr₂**, ..., **tr_n**; and
default values **dv₁**, **dv₂**, ..., **dv_n**.

In the tree example, the constructor is **buildTree**, a function of two arguments; the base is **emptyTree**; the recognizer is a predicate **isTree(...)**; and the accessors are **left**, **data**, and **right**. The type restrictions, which give the types of the results of the accessor functions, are **binaryTree**, **int**, **binaryTree**, respectively, and thus imply

```
left: [binaryTree → binaryTree]
data: [binaryTree → int]
right: [binaryTree → binaryTree]
```

Finally, the default values are **emptyTree**, 0, **emptyTree** for **left**, **data**, and **right**, respectively.

Just as before, models of structures introduced by the shell principle are obtained in either of two ways:

1. n -tuples in which the i -th term satisfies the type restriction tr_i ; or
2. all terms built up from **base** using the constructor **const**.

Axioms for the user-defined data type are automatically generated, a few of these being the following:³

$$\begin{aligned}
 r(x) &= T \vee r(x) = F \\
 r(\text{const}(x_1, \dots, x_n)) &= T \\
 r(\text{base}) &= T \\
 \text{base} &\neq \text{const}(x_1, \dots, x_n) \\
 r(x) &\rightarrow [x \neq \text{base} \rightarrow x = \text{const}(ac_1, \dots, ac_n(x))]
 \end{aligned}$$

Finiteness is expressed with the aid of an integer-valued function **count** and the axioms

$$\begin{aligned}
 \text{count}(\text{base}) &= 0 \\
 r(x) \ \& \ x \neq \text{base} &\rightarrow \text{count}(x) = 1 + \sum_{i=1}^n \text{count}(ac_i(x_i))
 \end{aligned}$$

The function **count** is not like the “**size**” and “**treeSize**” functions of earlier examples. It is special in that its definition enlarges as more data types (the “shells”) are defined, and it applies meaningfully to all defined data types; thus, if x_i above is replaced by an instance of some other defined data type with a different constructor, then **count** will return a value for that also.

The type restrictions in the shell principle may take either of two forms: a union of types (“one of”) or a complement of a union of types (“none of”). So the **const** function may be polymorphic. When **const** is applied to objects that do not meet the appropriate type restrictions, the appropriate default value is used instead. Likewise, the accessor functions return default values when applied to arguments not of the defined type.

³The use of r in some of these axioms is needed for a language in which variables can range over other objects as well as the type being defined; it could be omitted in a typed situation where x can be declared to have the type being defined.

The shell principle does not cover all conceivable user-defined data types that implement abstract data-type definitions. In particular, it does not allow for mutually recursive definitions, i.e., for a situation in which two new user-defined data types are being defined, with the constructor for each taking one or more arguments to be of the other type. Concrete examples where this actually needs to be done are rare, and it seems that the shell principle is adequate for practical cases. However, it would not be particularly difficult to extend the shell principle to accomodate multiple constructors.

2 SDVS Implementation

The shell principle suffices for the known cases of interest, so we have made it SDVS's paradigm for data type introduction [2]. We describe the necessary user interface by means of an example, defining the **stack** type:

```
<sdvs.1> createdatatype
datatype name: stack
  constructor: push
    arity: 2
  accessor#1: top
    accessor#1 type is stack [arbitrary]: integer
    accessor#1 default value: 0
  accessor#2: pop
    accessor#2 type is stack [arbitrary]: stack
    accessor#2 default value: emptyStack
  base: emptyStack
```

This is a stack of integers. It is possible to have stacks of other types also. The system requests a name for a base constant only if at least one of the accessors has been given a type that is the same as the one being defined—there is no need for a base constant in the case of nonrecursive data types.

Once the user has completed the input to the *createdatatype* command, SDVS creates the necessary new function symbols, in this case **top**, **push**, **emptystack**, and **stacksize**. The last is not mentioned by the user, but it is automatically supplied by SDVS; it is intended to return an integer measure of the size of objects of the new type. Next, SDVS introduces a set of new axioms for the functions and constants just introduced:

Datatype 'stack' created with the following axioms:

```
axiom stack.1 (i,s): emptystack ~= push(i,s)

axiom stack.2 (s): emptystack ~= s --> s = push(top(s),pop(s))

axiom stack.3 (i,s): top(push(i,s)) = i

axiom stack.4 (i,s): pop(push(i,s)) = s
```

```
axiom stack.5 (): stacksize(emptystack) = 0
```

```
axiom stack.6 (i,s): stacksize(push(i,s)) = 1 + stacksize(s)
```

Inductive proofs about stacks can be done with the aid of the `stacksize` function.

We have presented the framework for the user interface to the new features of SDVS entirely by means of an example: the case of the `stack` data type. There are no essential differences between this and the general case. Sometimes the general situation can be simpler: the `base` constant, the `size` function, and the structural induction rule are introduced only in case the defined data type is recursive.

3 Conclusions

Our survey of the problem of abstract data type definitions and their requirements has indicated that the Boyer-Moore shell principle is adequate for most practical purposes. Accordingly, we have incorporated it into SDVS.

There are possible extensions to the definition facility that might be made, most notably one that allows mutually recursively defined data types. Further facilities could be introduced into SDVS, especially induction rules specially designed for new data types. Neither type of extension appears necessary at the present stage of development.

References

- [1] R. S. Boyer and J. S. Moore, *The User's Manual for a Computational Logic*. Computation Logic, Inc., 1987.
- [2] L. Marcus, "SDVS 8 Users' Manual," Tech. Rep. ATR-89(4778)-4, The Aerospace Corporation, Sept. 1989.